COMP1511: Self Referential Structures - Linked List

Session 2, 2018

Recap: Self-Referential Structures

We can define a structure containing
a pointer to the same type of structure:

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

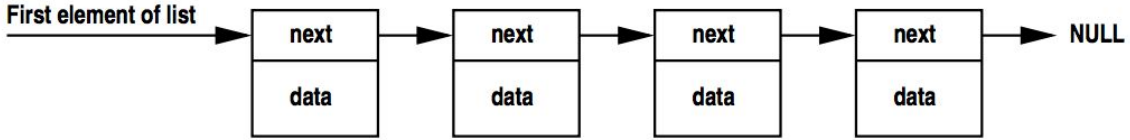
These "self-referential" pointers can be used to build larger "dynamic" data structures out of smaller building blocks.

Recap: Linked List

The most fundamental of these dynamic data structures is the *Linked List*:

- based on the idea of a sequence of data items or nodes
- linked lists are more flexible than arrays:
  - items don't have to be located next to each other in memory
  - items can easily be rearranged by altering pointers
  - the number of items can change dynamically
  - items can be added or removed in any order

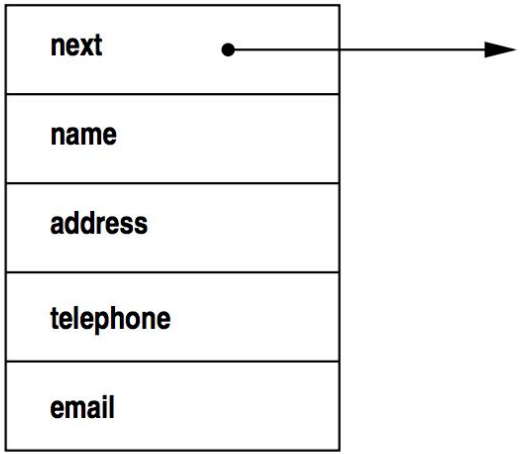
Recap: Linked List



- a *linked list* is a sequence of items
- each item contains data and a pointer to the next item
- need to separately store a pointer to the first item or "head" of the list
- the last item in the list is special
  it contains `NULL` in its `next` field instead of a pointer to an item

Recap: Example of List Item

Example of a list item used to store an address:



Recap: Example of List Item in C

```
struct address_node {
    struct address_node *next;
    char *telephone;
    char *email;
    char *address;
    char *telephone;
    char *email;
};
```

Recap: List Items

List items may hold large amount of data or many fields.
For simplicity, we'll assume each list item need store only a single int.

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

Recap: List Operations

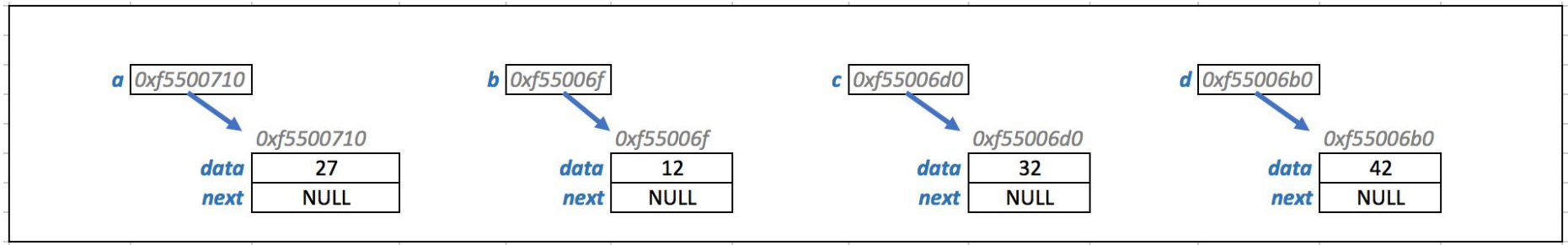
Basic list operations:

- create a new item with specified data
- search for a item with particular data
- insert a new item to the list
- remove a item from the list

Many other operations are possible.

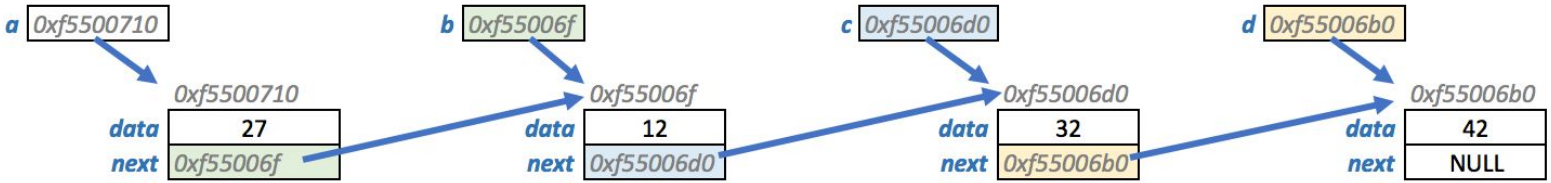
Creating a Node (List Item)

```
struct node *a = malloc(sizeof (struct node));
a->data = 27;
a->next = NULL;

struct node *b = malloc(sizeof (struct node));
b->data = 12;
b->next = NULL;

struct node *c = malloc(sizeof (struct node));
c->data = 32;
c->next = NULL;

struct node *d = malloc(sizeof (struct node));
d->data = 42;
d->next = NULL;
```

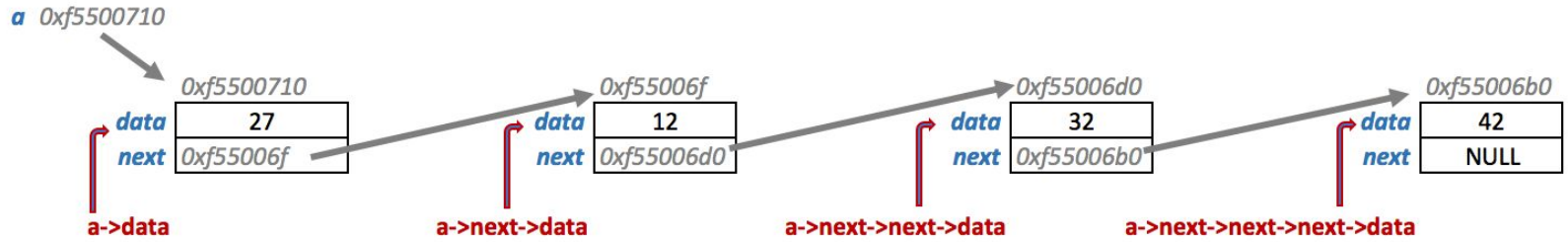


Link Nodes

```
a->next = b;
b->next = c;
c->next = d;
d->next= NULL;
```

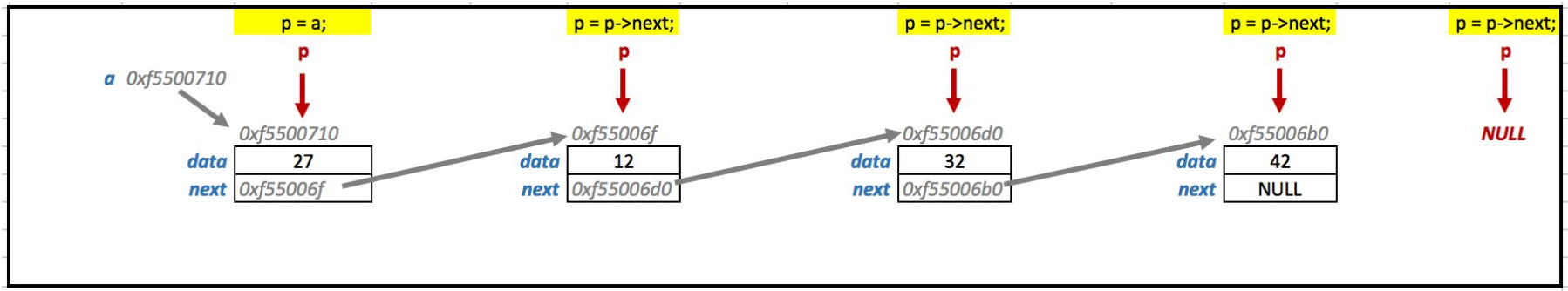


Link list




```
printf("---------------- ----------------\n\n");
printf("// After linking nodes: \n");
printf("//    a->next is same as b\n");
printf("//    a->next->next is same as c\n");
printf("//    a->next->next->next is same as d\n");
printf("   Node a->data: %d \n", a->data);
printf("   Node a->next->data: %d \n", a->next->data);
printf("   Node a->next->next->data: %d \n", a->next->next->data);
printf("   Node a->next->next->next->data: %d \n", a->next->next->next->data);
```

Link List - Traversal

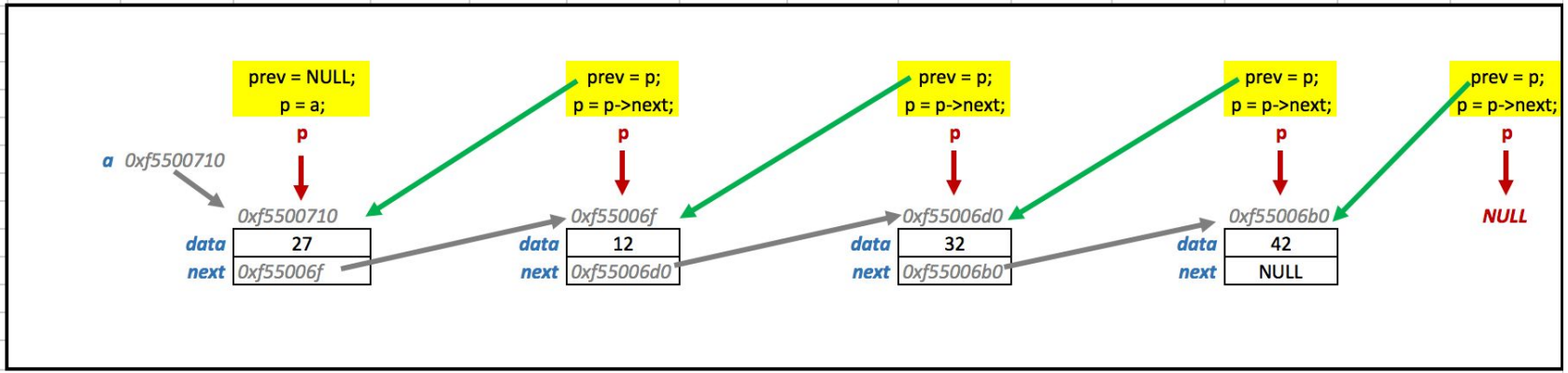



```
printf("---------------- ----------------\n\n");
printf("//  Below: process_list(a) ..... \n\n");
process_list(a);
```

```
void process_list(struct node *head) {
    struct node *p = head;

    while (p != NULL) {

        printf("    p->data=%d \n", p->data );

        p = p->next;
    }
}
```

Process node data here

Linked List: Previous pattern



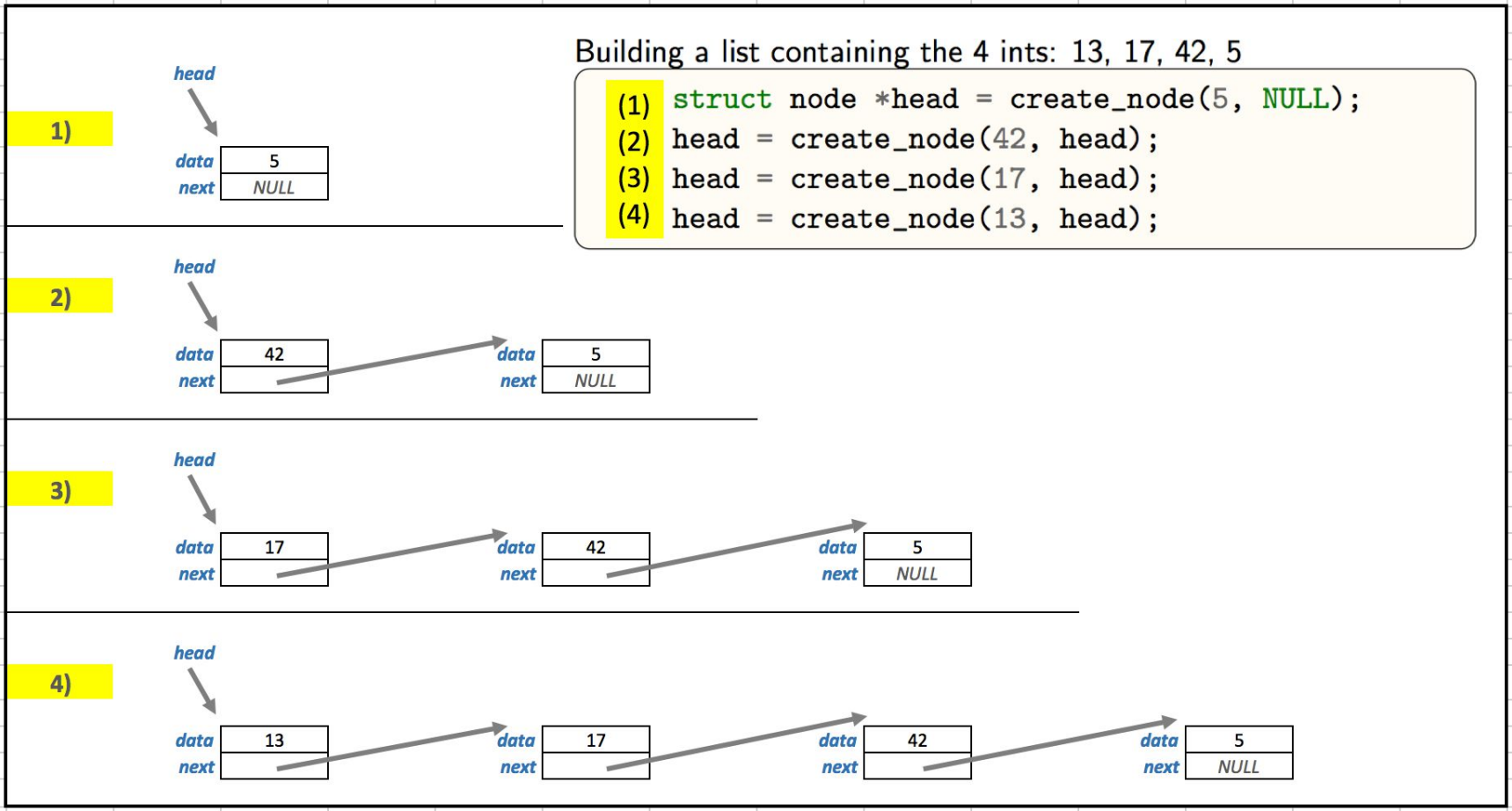

```c
void prev_example(struct node *head) {
    struct node *prev = NULL;
    struct node *p = head;

    while (p != NULL) {

        printf("   p->data=%d \n", p->data );

        prev = p;
        p = p->next;
    }
}
```

Later we will see examples of this **previous pattern**, for example in **deleting** a node.

Creating a List Item/Node

```c
// Create a new struct node containing the specified dat
// and next fields, return a pointer to the new struct n

struct node *create_node(int data, struct node *next) {
    struct node *n;
    n = malloc(sizeof (struct node));
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    n->data = data;
    n->next = next;
    return n;
}
```

Building a list



Summing a List

```c
// return sum of list data fields
int sum(struct node *head) {
    int sum = 0;
    struct node *n = head;
    // execute until end of list
    while (n != NULL) {
        sum += n->data;
        // make n point to next item
        n = n->next;
    }
    return sum;
}
```

Summing a List: For Loop

```c
// return sum of list data fields: using for loop

int sum1(struct node *head) {
    int sum = 0;

    for (struct node *n = head; n != NULL; n = n->next) {

        sum += n->data;

    }

    return sum;
}
```

Finding an Item in a List

```c
// return pointer to first node with specified data value
// return NULL if no such node

struct node *find_node(struct node *head, int data) {
    struct node *n = head;

    // search until end of list reached
    while (n != NULL) {
        // if matching item found return it
        if (n->data == data) {
            return n;
        }

        // make node point to next item
        n = n->next;
    }

    // item not in list
    return NULL;
}
```

Finding an Item in a List: For Loop

- Same function but using a for loop instead of a while loop.
- Compiler will produce same machine code as previous function.

```
// previous function written as for loop

struct node *find_node1(struct node *head, int data) {
    for (struct node *n = head; n != NULL; n = n->next) {
        if (n->data == data) {
            return n;
        }
    }
    return NULL;
}
```

Finding an Item in a List: Shorter While Loop

- Same function but using a more concise while loop.
- Shorter does not always mean more readable.
- Compiler will produce same machine code as previous functions.

```
struct node *find_node2(struct node *head, int data) {
    struct node *n = head;

    while (n != NULL && n->data != data) {

        n = n->next;
    }

    return n;
}
```

Printing a List - Python Syntax

For example,
[45, 67, 2, 43]

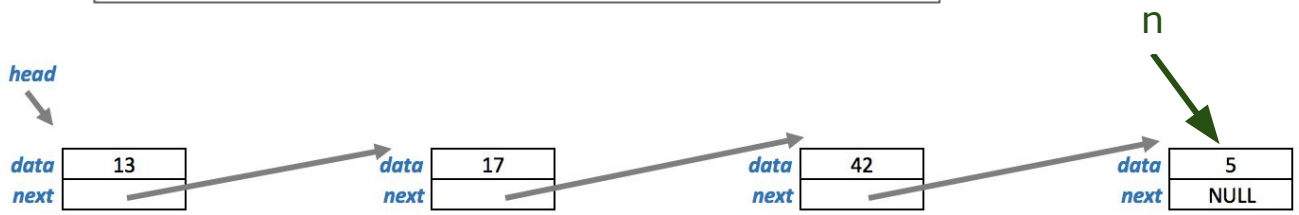
```
// print contents of list in Python syntax

void print_list(struct node *head) {
    printf("[");
    for (struct node *n = head; n != NULL; n = n->next) {
        printf("%d", n->data);
        if (n->next != NULL) {
            printf(", ");
        }
    }
    printf("]");
}
```

We print “,” if there is a next node, otherwise skip printing “,”

Finding Last Item in List

```
// return pointer to last node in list
// NULL is returned if list is empty

struct node *last(struct node *head) {
    if (head == NULL) {
        return NULL;
    }

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

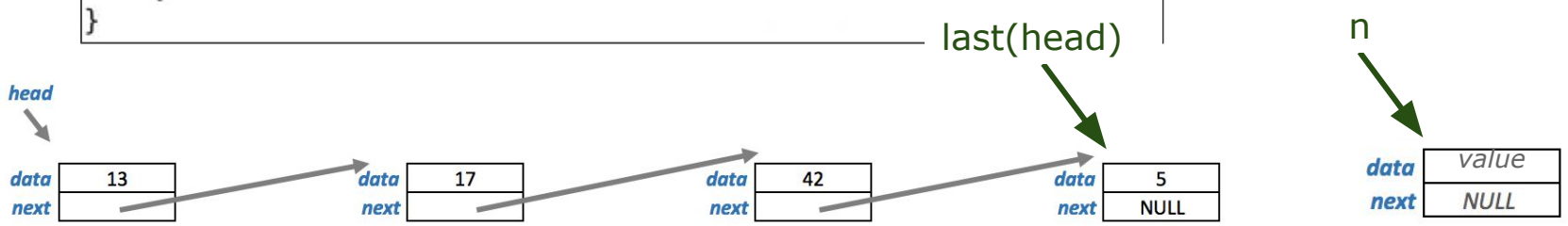
See the difference:
We are checking,
n->next != NULL
(in place of n != NULL)



The loop stops here because,
n->next == NULL

Appending to List

```c
// create a new list node containing value
// and append it to end of list

struct node *append(struct node *head, int value) {
    // new node will be last in list, so next field is NULL
    struct node *n = create_node(value, NULL);
    if (head == NULL) {
        // new node is now  head of the list
        return n;
    } else {
        // change next field of last list node
        // from NULL to new node
        last(head)->next = n;  /* append node to list */
        return head;
    }
}
```



Deleting all items from a List

```c
// Delete all the items from a linked list.

void delete_all(struct node *head) {
    struct node *n = head;
    struct node *tmp;

    while (n != NULL) {
        tmp = n;
        n = n->next;
        free(tmp);
    }
}
```

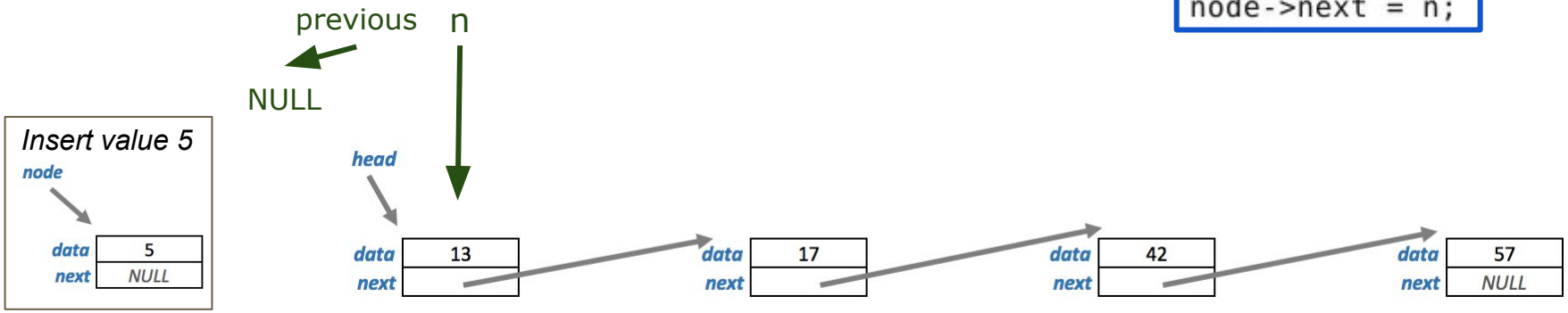
We **cannot** do the following in the body of while loop!

```c
free(n);
n = n->next;
```

Insert a Node into an Ordered List

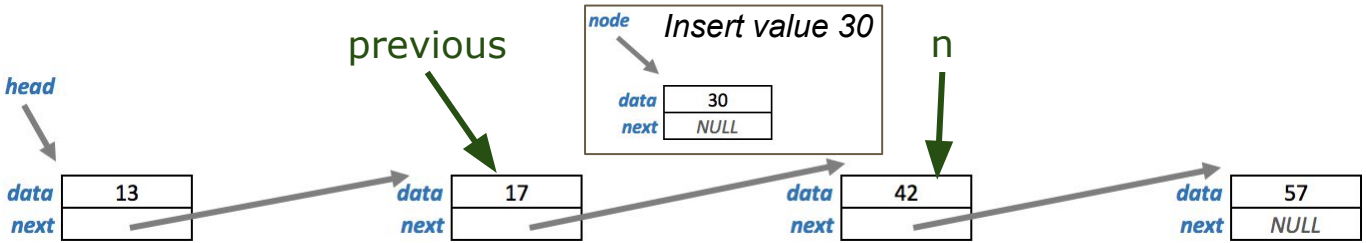
- Case-1: Insert a node at the beginning

```
head = node;
node->next = n;
```



- Case-2: Insert a node in the middle or at the end

```
previous->next = node;
node->next = n;
```



Insert a Node into an Ordered List

```
//Insert a Node into an Ordered List
//
struct node *insert_ordered(struct node *head, struct node *node) {
    struct node *previous;
    struct node *n = head;
    // find correct position
    while (n != NULL && node->data > n->data) {
        previous = n;
        n = n->next;
    }

    // link new node into list
    if (previous == NULL) {
        head = node;
        node->next = n;
    } else {
        previous->next = node;
        node->next = n;
    }

    return head;
}
```

Find correct position

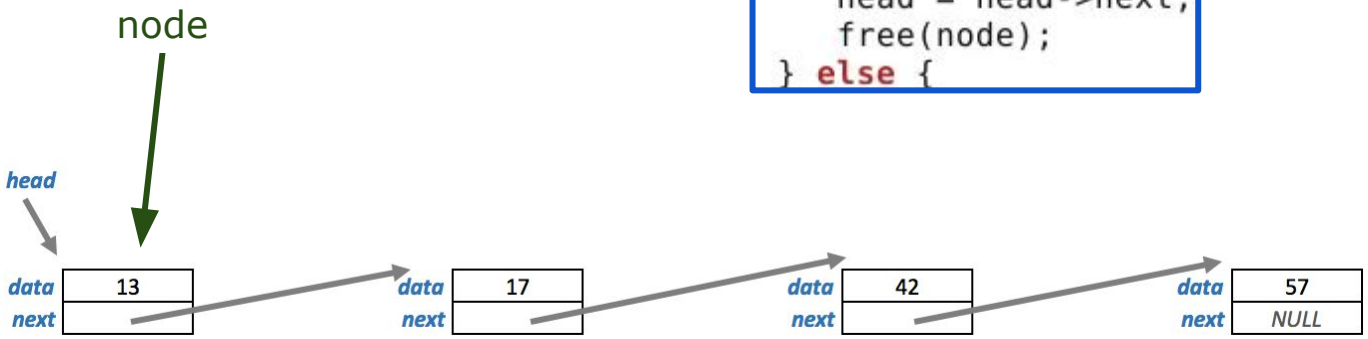
Case-1

Case-2

Delete a Node from a List

- Case-1: Remove first node

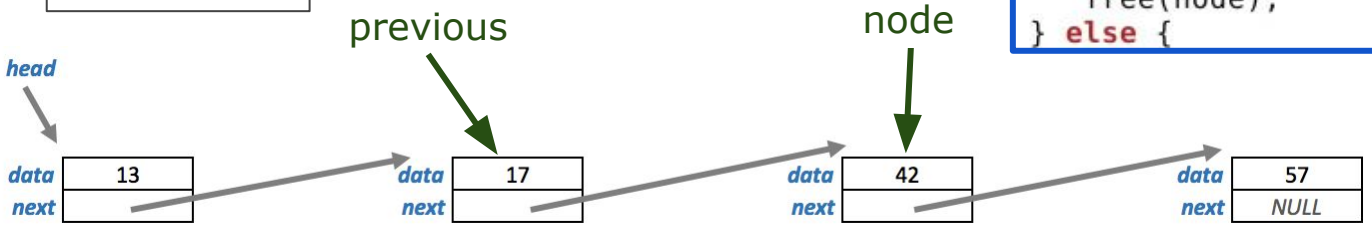
Delete node with value “13”




```
if (node == head) {
    head = head->next;
    free(node);
} else {
```

- Case-2: Remove a node in the middle or at the end

Delete node with value “42”




```
if (previous != NULL) { // node found in list
    previous->next = node->next;
    free(node);
} else {
```

Delete a Node from a List

```c
// Delete a Node from a List
//
struct node *delete(struct node *head, struct node *node) {
   if (node == head) {
      head = head->next;          // remove first item
      free(node);
   } else {
      struct node *previous = head;
      while (previous != NULL && previous->next != node) {
         previous = previous->next;
      }
      if (previous != NULL) { // node found in list
         previous->next = node->next;
         free(node);
      } else {
         fprintf(stderr, "warning: node not in list\n");
      }
   }
   return head;
}
```

Case-1

Find correct position

Case-2