5. Data Structures

2023-12-18

Section materials: jrsacher.github.io/cs50/

Agenda

- The last week of C!
- Quick review of structs
- Linked lists
  - Nodes
  - `->` operator
  - Comparison with arrays
- Stacks, queues
- Hash tables
- Trees
- Tries

typedef and structs

```
typedef old-name new-name;
typedef char * string;
typedef struct car
{
   int year;
   char model[10];
   char plate[7];
   bool automatic;
}
car;
```

Accessing structs

```
// declaration
car mycar;

// field access
mycar.year = 2017;
strcpy(mycar.plate, “CS50”);
mycar.automatic = true;
```

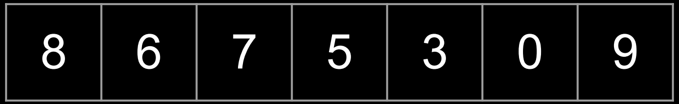
Accessing structs (with pointers!)

```
// declaration
car *mycar = malloc(sizeof(car));

// field access
mycar->year = 2017;
strcpy(mycar->plate, “CS50”);
mycar->automatic = true;
// equivalent to (*mycar).automatic = false
```

linked lists

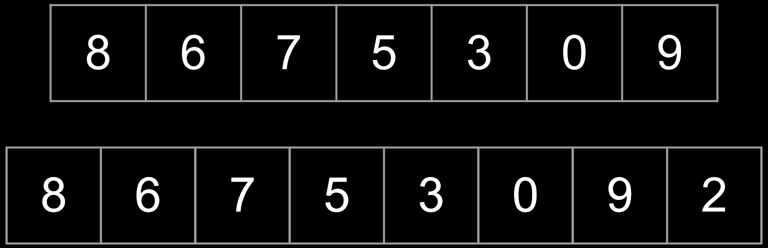
Arrays

Data stored sequentially in memory



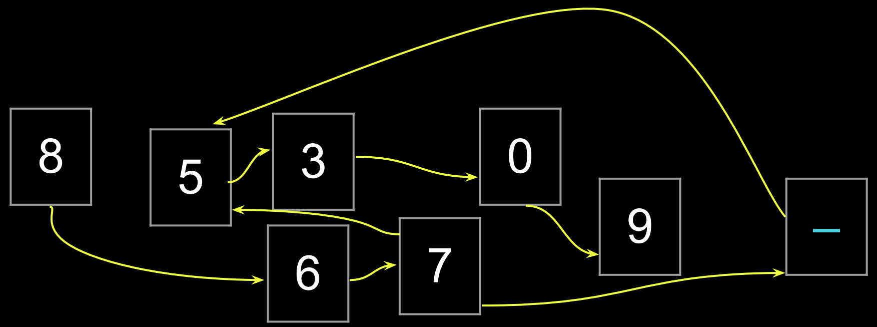
- Good
  - Rapid, random access
  - Efficient storage of data
- Bad
  - Fixed size

"Growing" arrays



Linked lists

Data stored separately, but connected



- Good
  - Shrink and grow as needed
  - Can insert or delete in arbitrary order
- Bad
  - No random access – linear search
    - There are *technically* ways around this, but they get complicated.
  - Takes more memory to store – the pointer takes extra space

Linked Lists

- A linked list **node** is a special type of `struct` with (minimally) 2 properties:
    - Data of some type
    - A pointer to another node in the linked list.

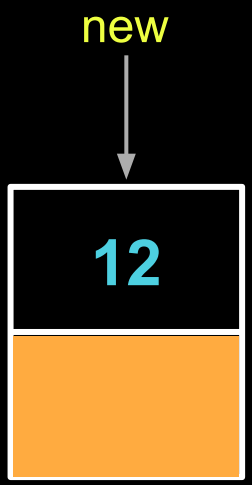
```
typedef struct node
{
    int value;
    struct node *next;
}
node;
```

Operations needed to work with linked lists

- Creating a linked list when it doesn’t exist.
- Inserting a new node into a linked list.
- Searching through a linked list to find an element.
- Deleting a single element from a linked list.
- Deleting an entire linked list.

Making a new linked list

- Dynamically allocate space for a new node with malloc.
  - This returns a pointer to your newly created node.
- Make sure you didn't run out of memory.
  - Check that the pointer isn't NULL
- Initialize the value field.
- Initialize the next field (specifically, to NULL).



In code …

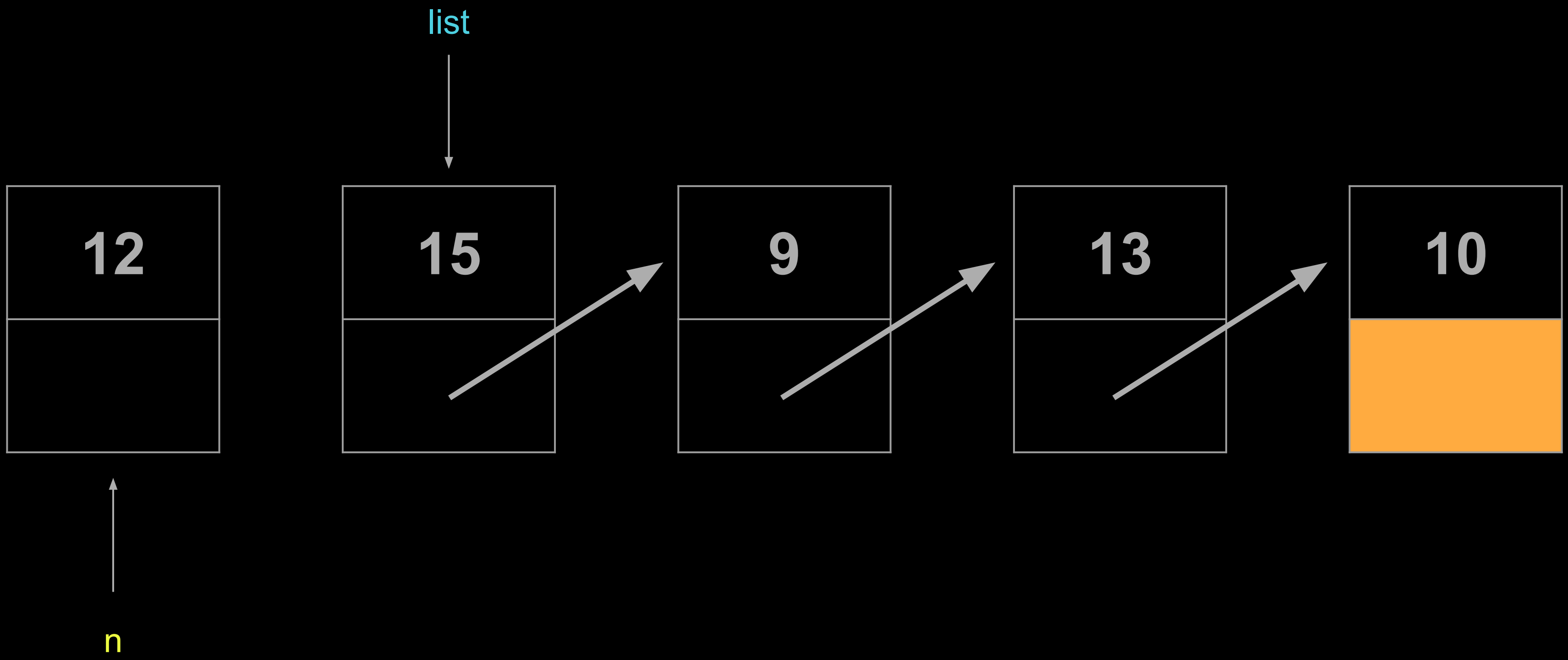
```
typedef struct node
{
  // the value to store in this node
  int value;
  // the link to the next node in the list
  struct node *next;
} node;

node *list = malloc(sizeof(node));
// Check that malloc succeeded
if (list == NULL)
{
   return 1;
}
// Add a value to the new node
list->value = 1;
list->next = NULL;
```

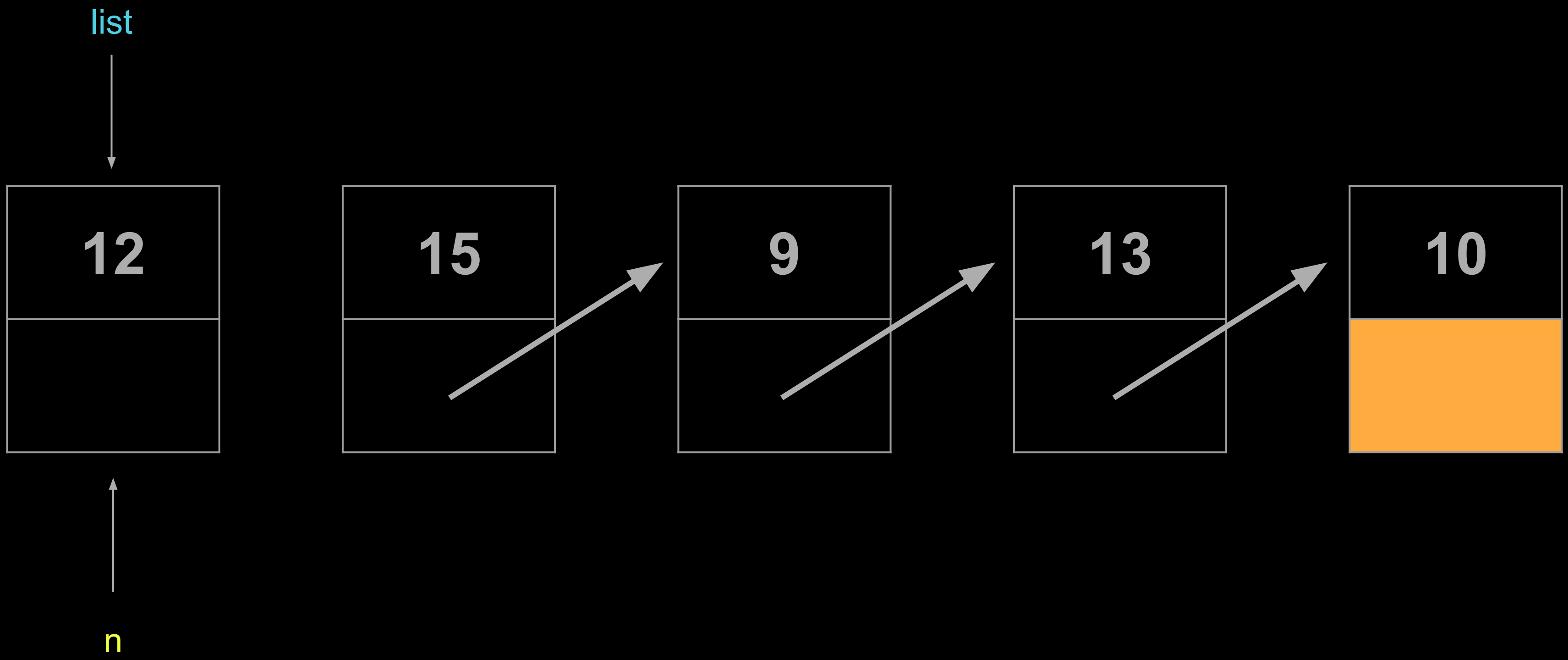
Inserting

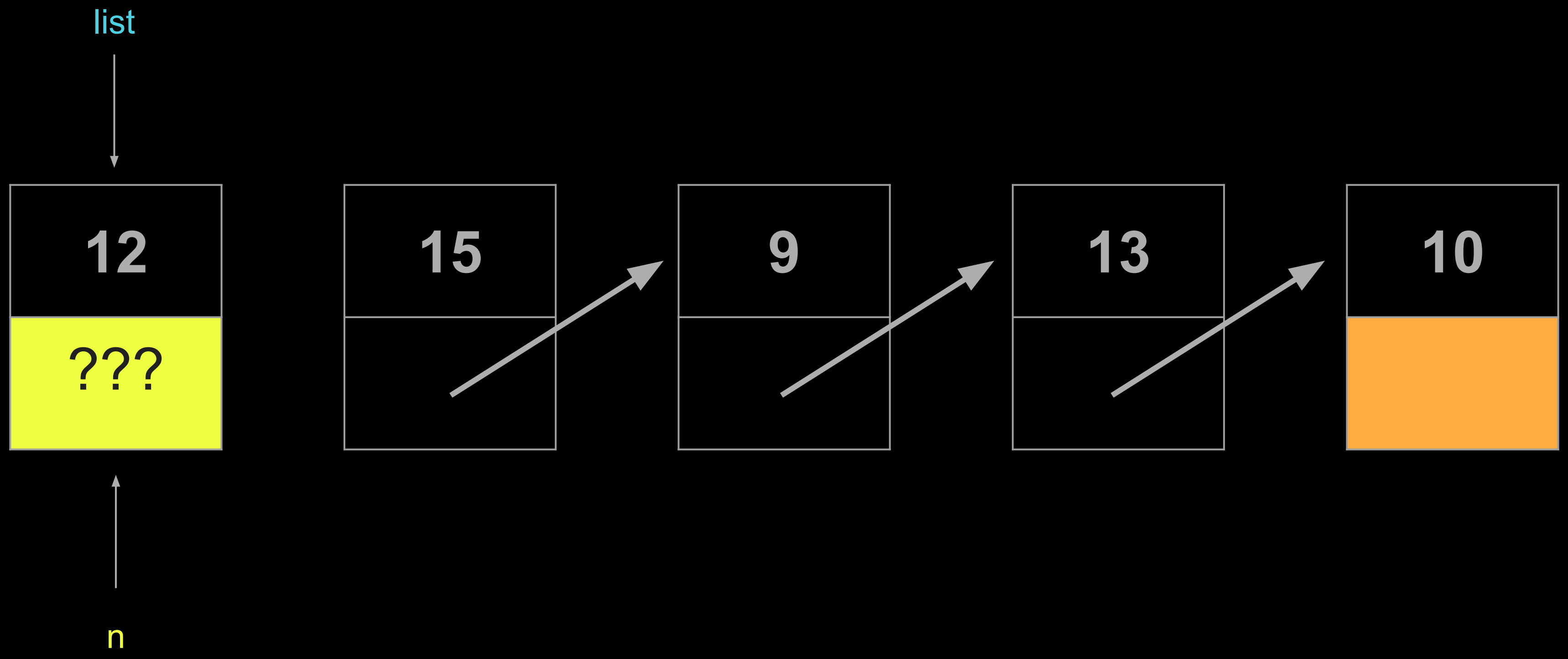
- Dynamically allocate space for a new linked list node.
  - Check to make sure we didn't run out of memory.
- Populate, insert node at the beginning of the linked list.
  - So which pointer do we move first? The pointer in the newly created node, or the pointer pointing to the *original* head of the linked list?
  - This choice matters!
- Return a pointer to the new head of the linked list.
- Note: can also insert into the middle or end of a linked list

In code ...

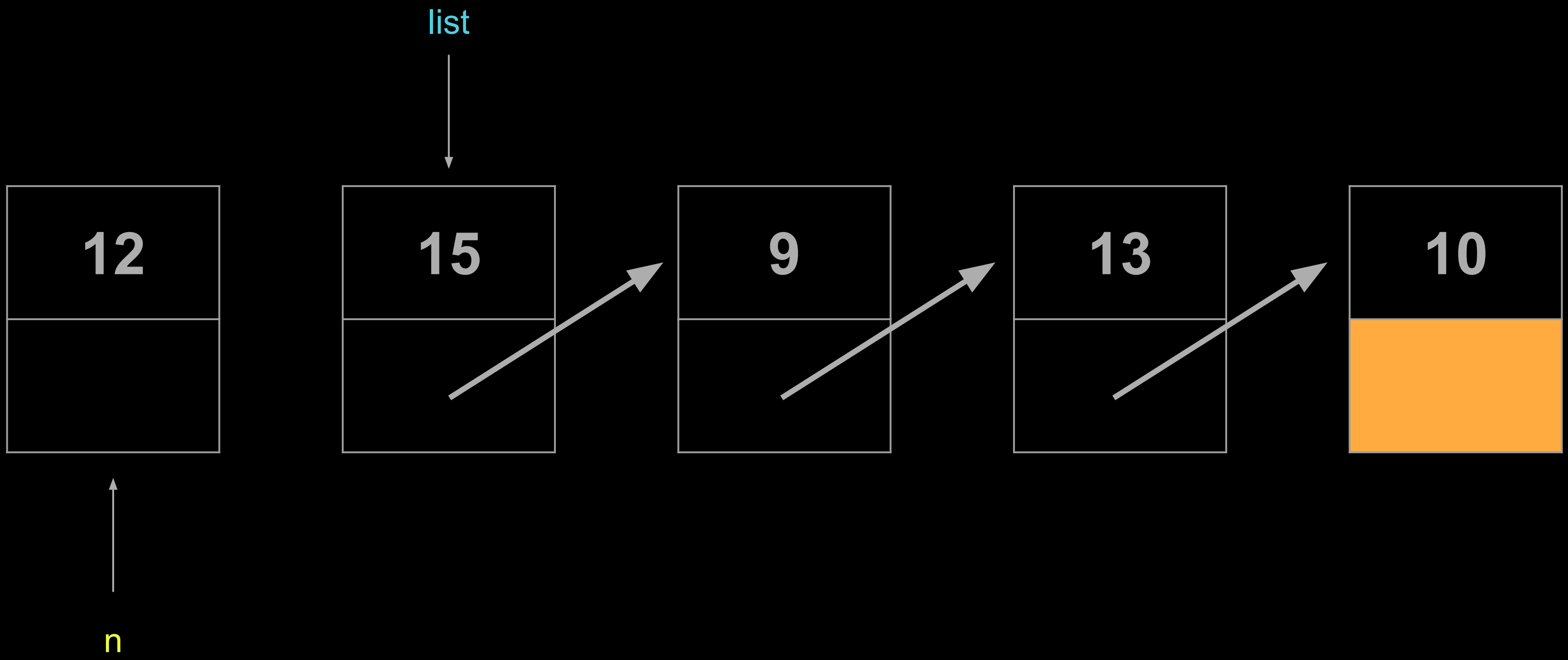
```c
// Create a new node
node *n = malloc(sizeof(node));
// Check that malloc succeeded
if (n == NULL)
{
   free_linked_list(list);
   return 1;
}
// Add a value to the new node
n->value = 2;
// Link the new node to the list
n->next = list;
// move the list to point to the new node
list = n;
list = n;
```



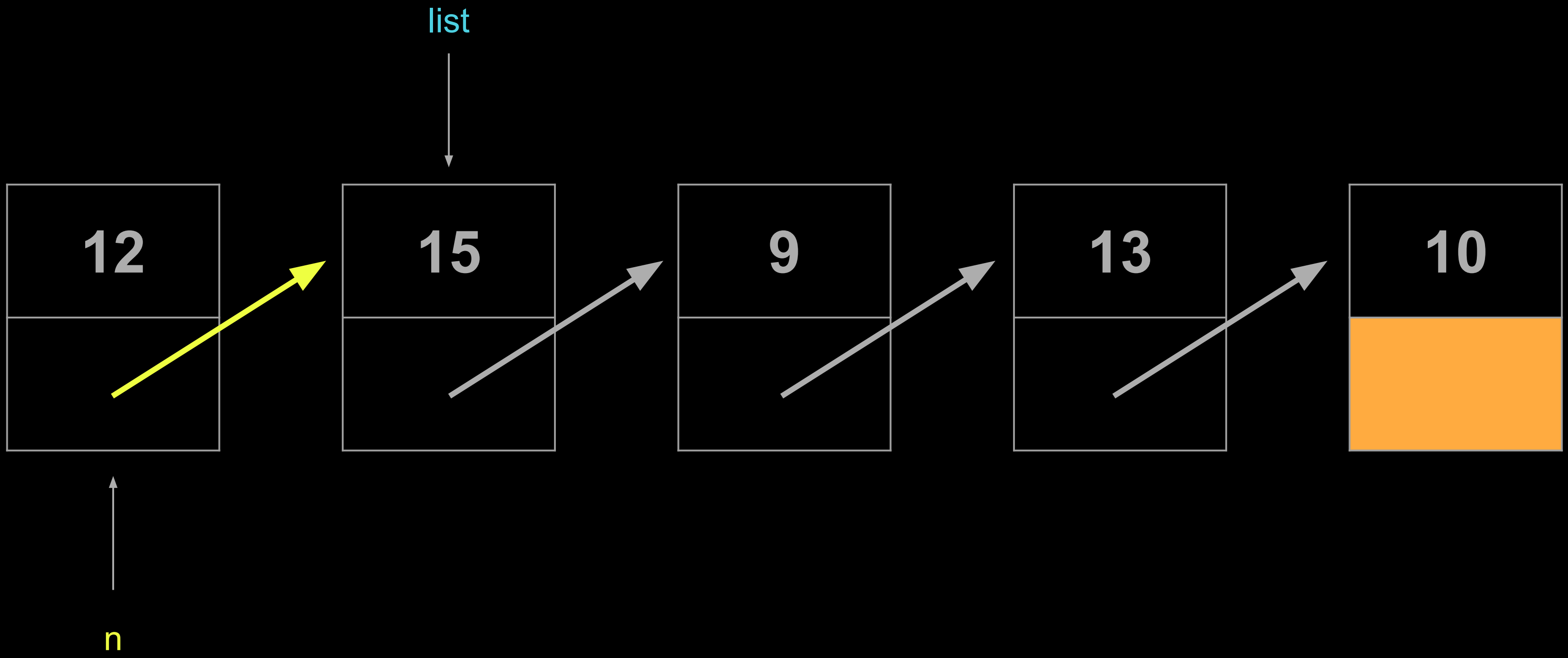

list
12
15
9
13
10
n


list
12
???
n
15
9
13
10

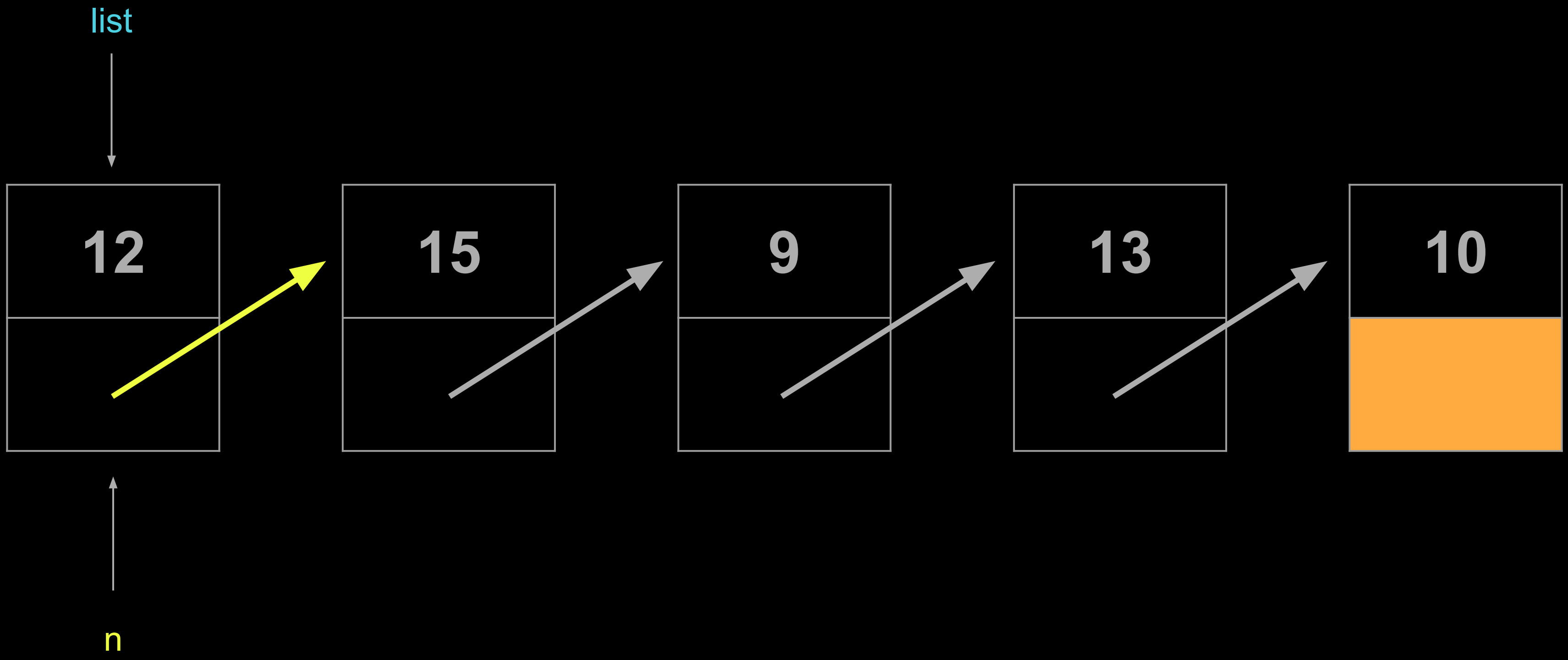
X


list
12
15
9
13
10
n

```
n->next = list;
```



```
list = n;
```



✓

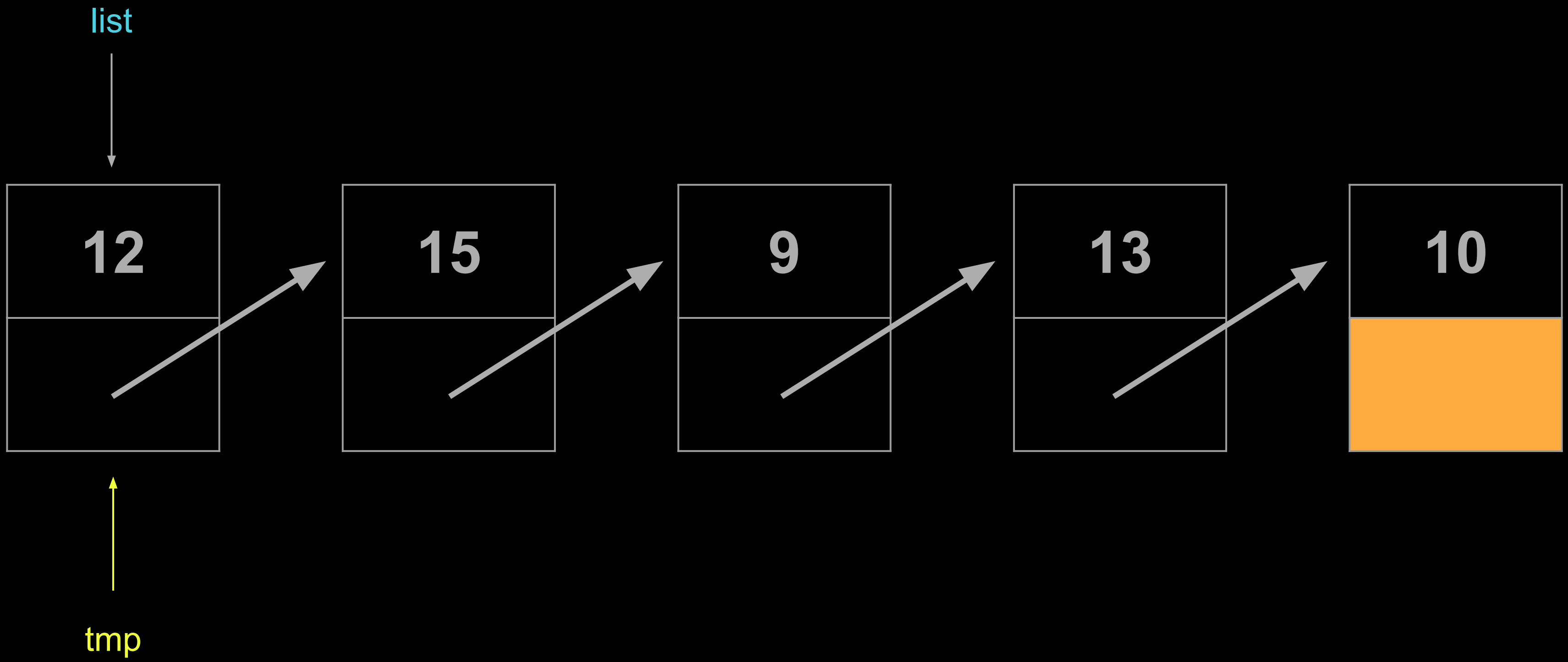
Searching

- Make a traversal pointer to list’s head (first element).
- If current node’s value field is what we want, return `true`.
- If not, set the traversal pointer to the next pointer in the list and go back to the previous step.
- If you’ve reached the end of the list, return `false`.

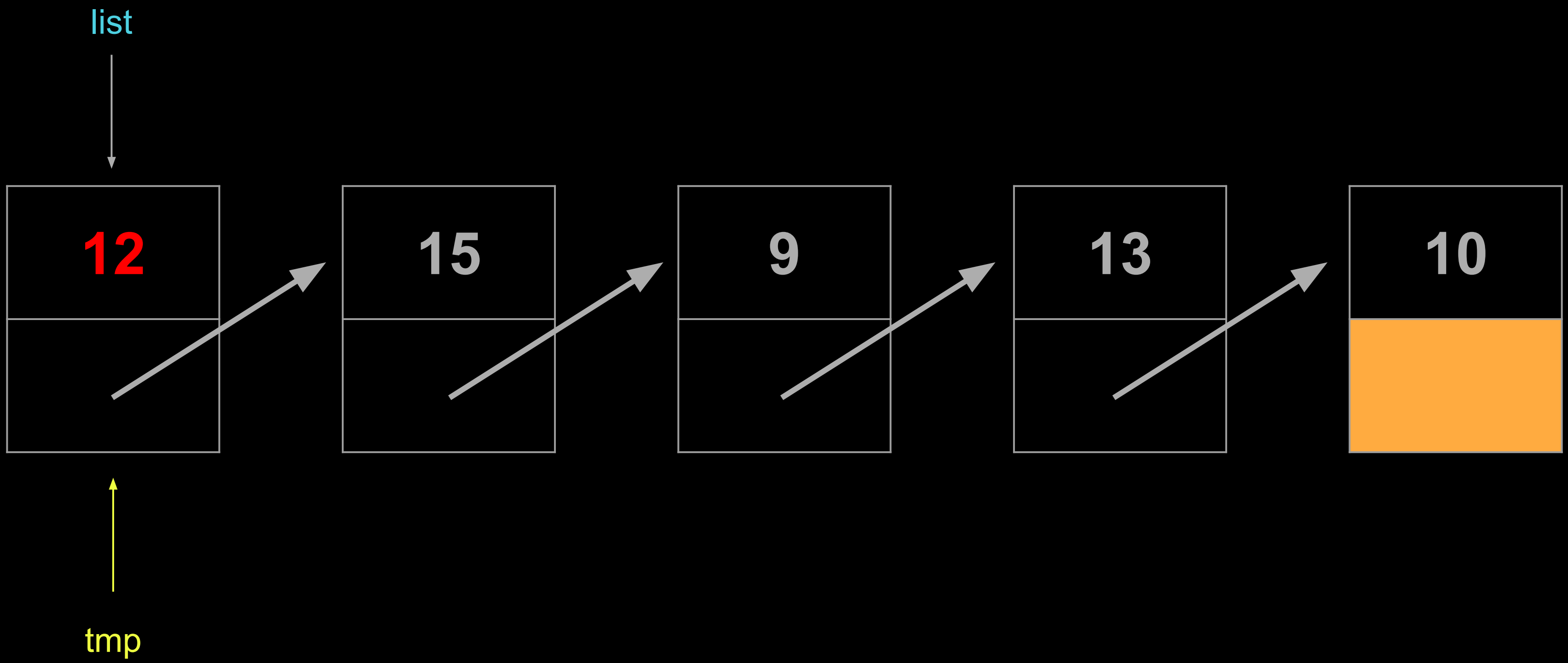
In code ...

```c
bool found = false;
node *tmp = list;
while (tmp != NULL && !found)
{
    if (tmp->value == n)
    {
        printf("Found\n");
        found = true;
        break;
    }
    tmp = tmp->next;
}
if (!found)
{
   printf("Not found\n");
}
```

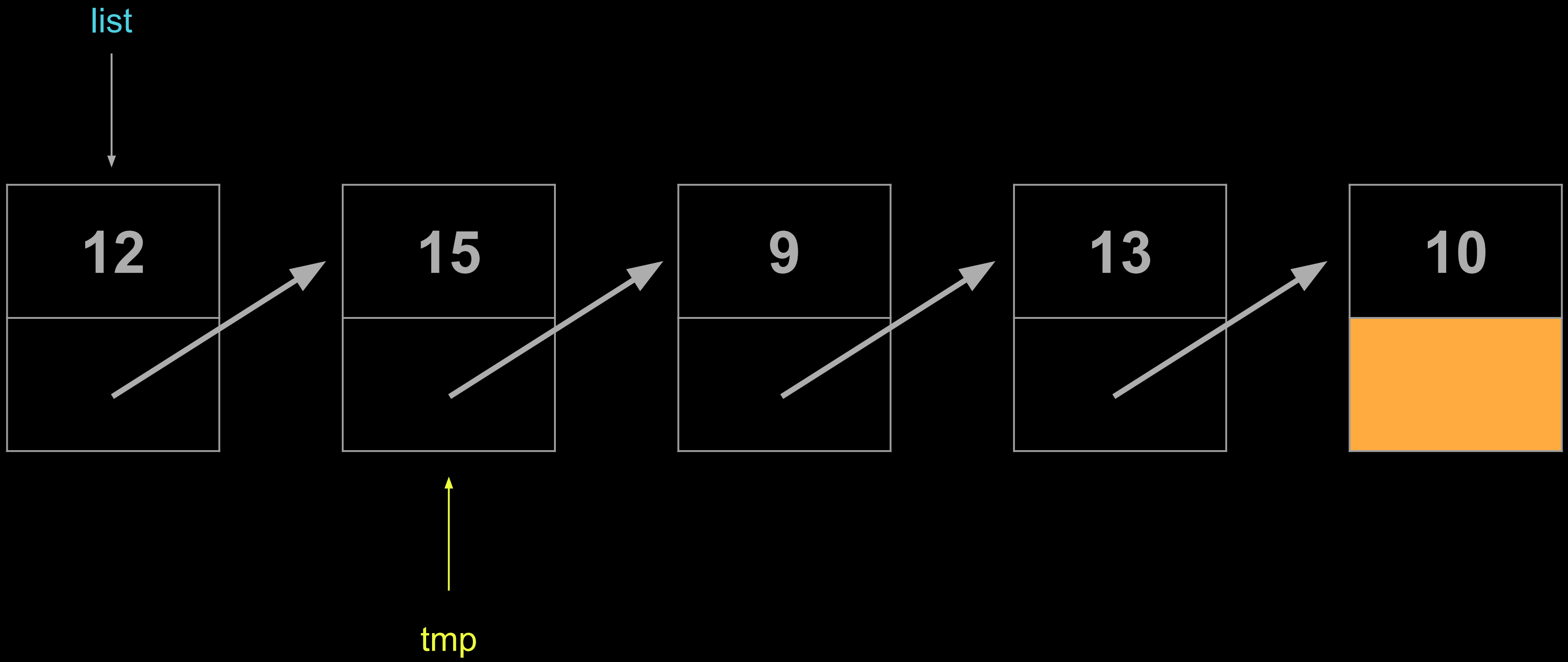
```
node *tmp = list;
```



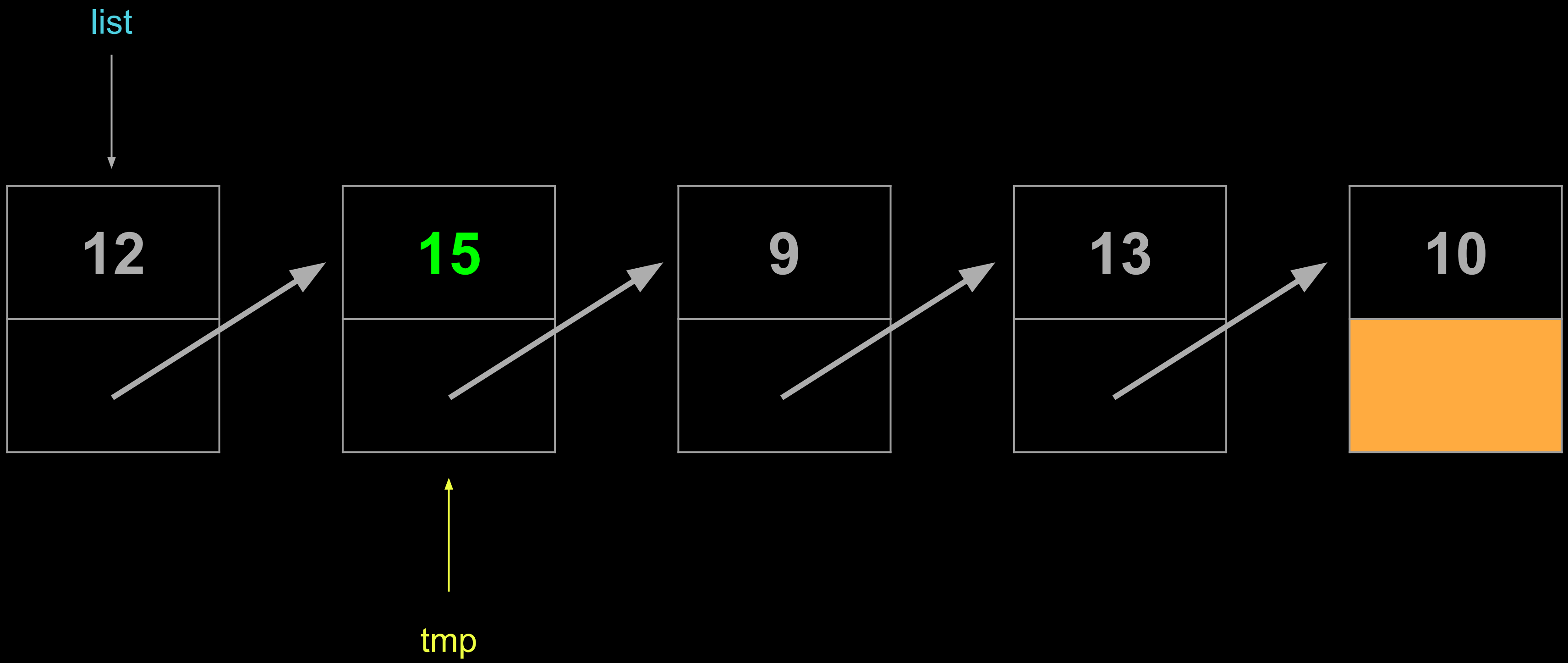
```
tmp->value == 15;
```



```
tmp = tmp->next;
```



```
tmp->value == 15;
```



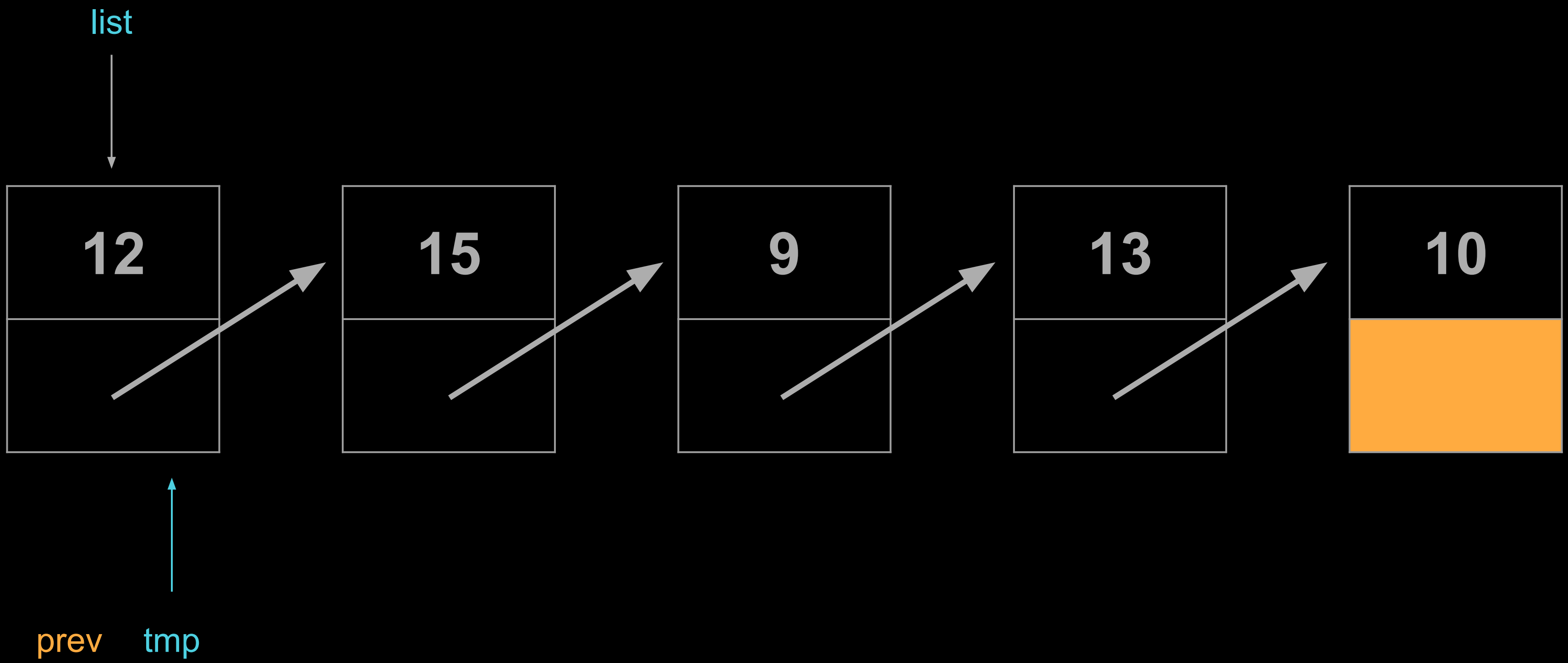
Deleting

- A bit like a combination of searching and inserting
- If deleting from the middle, use 2 temp pointers to traverse linked list
  - Keep track of current and previous node
- When you find the value to delete
  - Point the previous node to the one pointed to by the current node to close the gap
  - `free` the current node

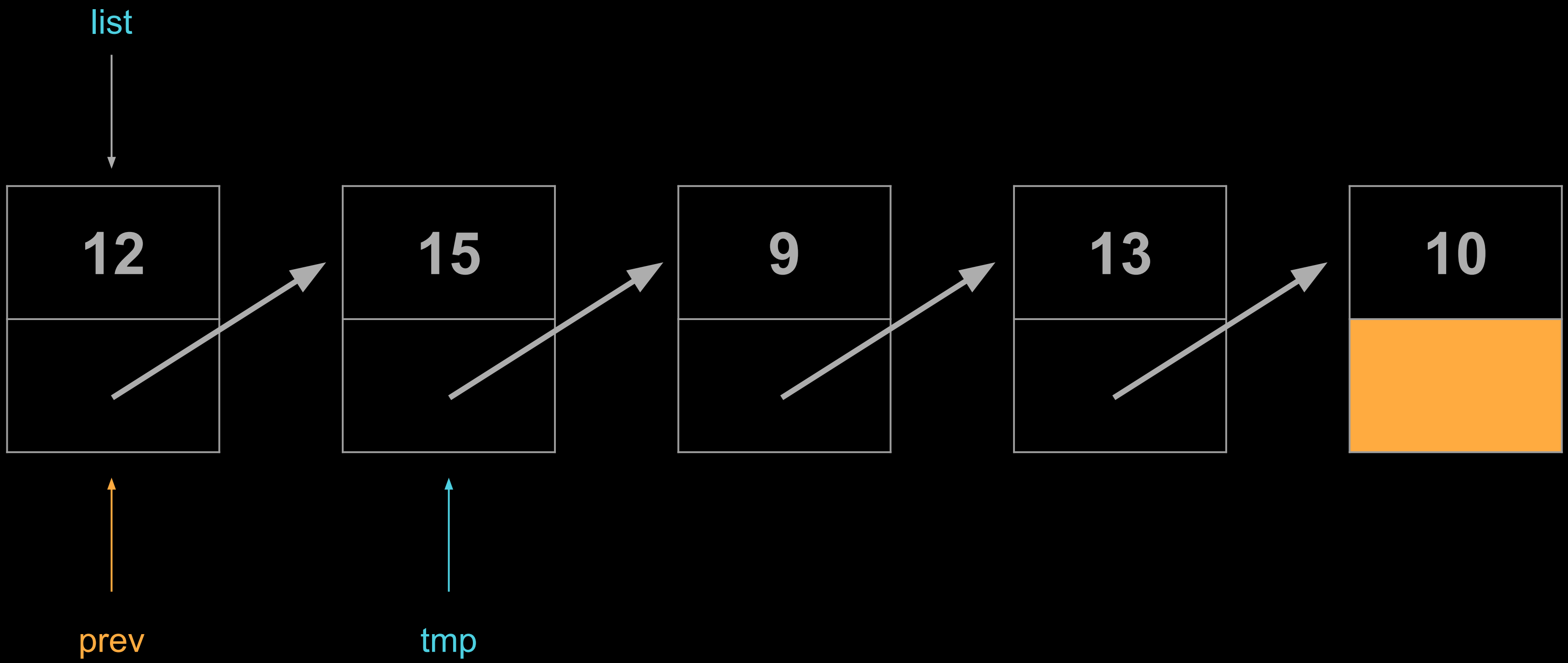
In code ...

```
node *tmp = list;
node *prev = NULL;
while (tmp != NULL)
{
    if (tmp->value == n)
    {
        if (prev == NULL)
        {
            // we need to delete the first node
            list = tmp->next;
        }
        else
        {
            // we need to delete some other node
            prev->next = tmp->next;
        }
        free(tmp);
        break;
    }
    prev = tmp;
    tmp = tmp->next;
}
```

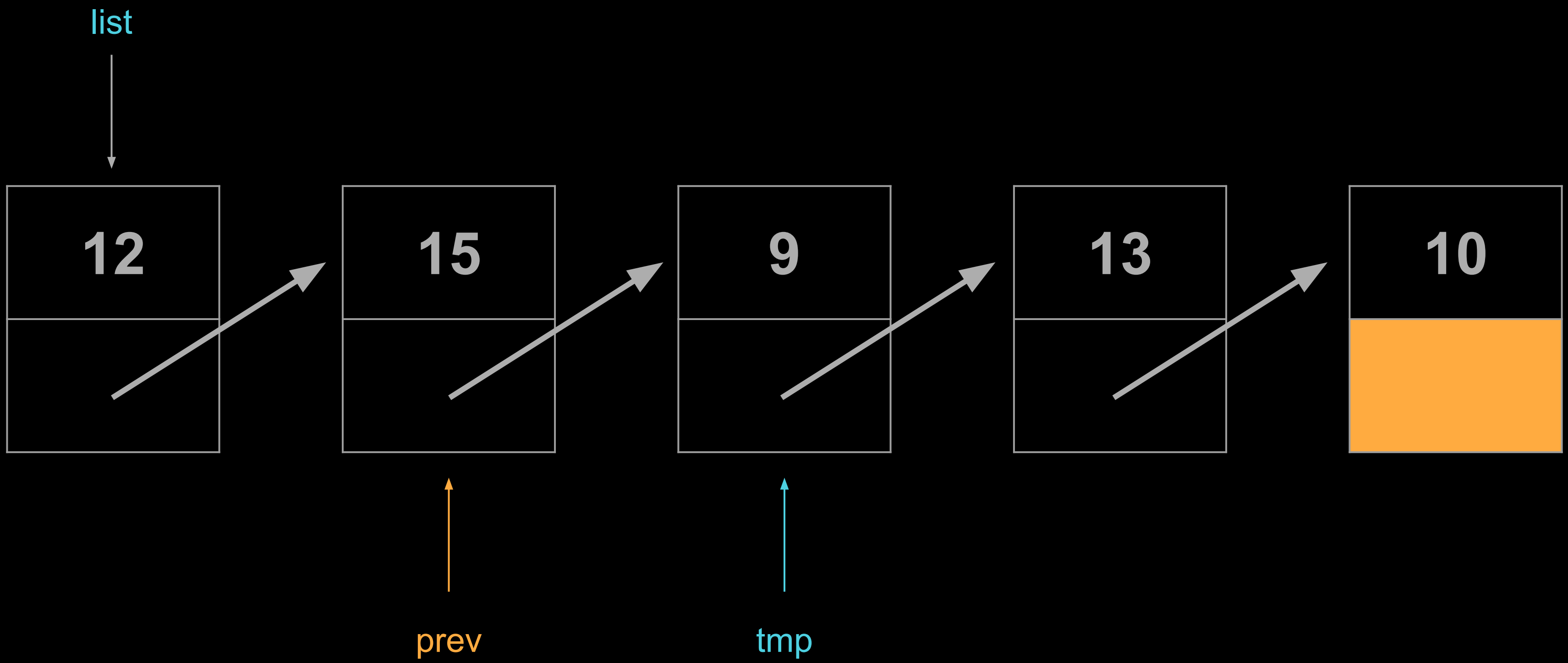
node *tmp = list; node *prev = NULL;



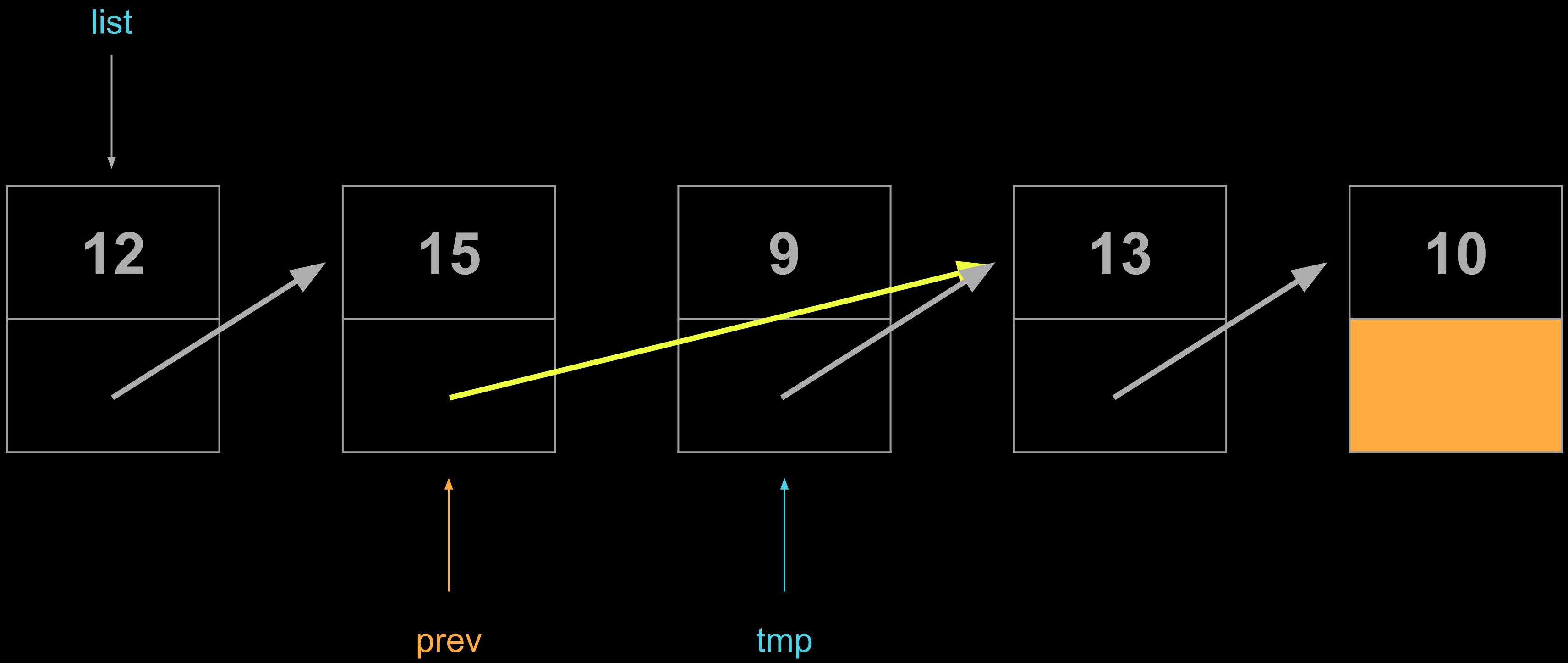
```
prev = tmp; tmp = tmp->next;
```



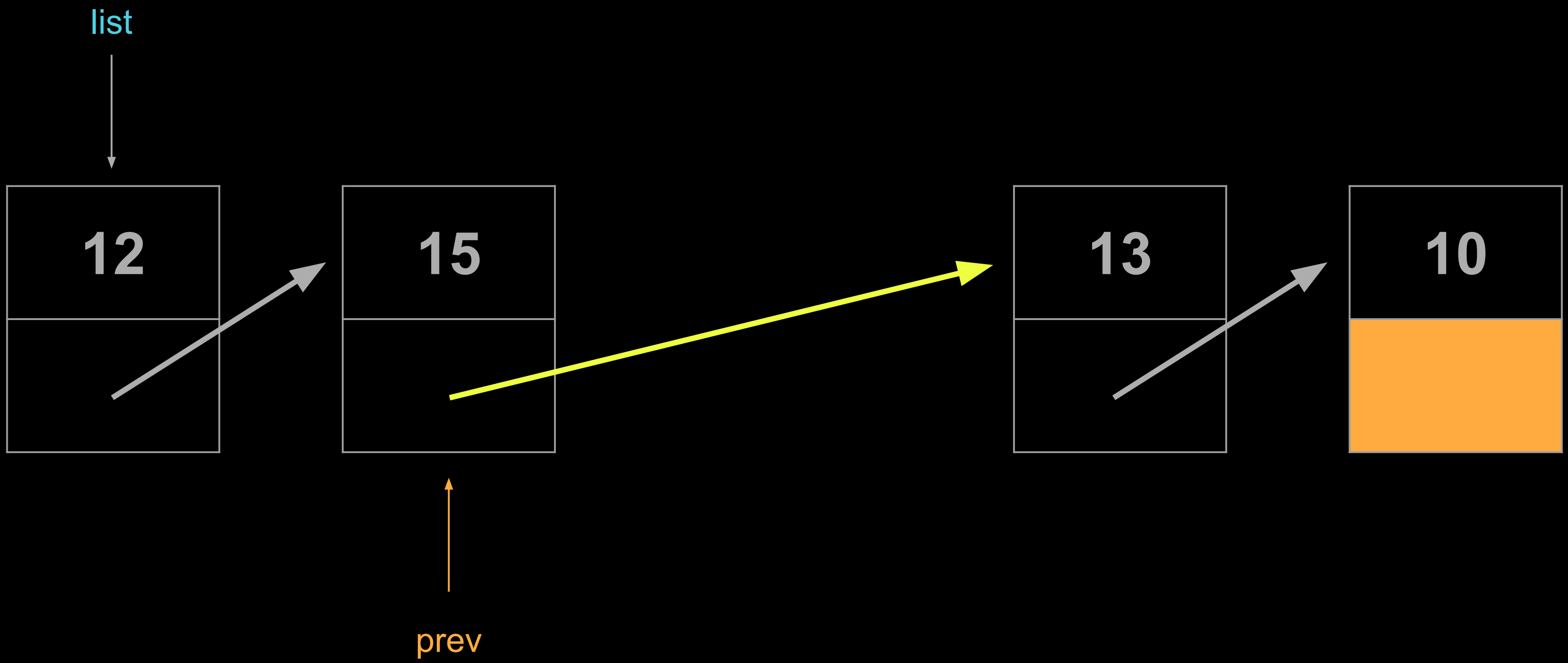
```
prev = tmp; tmp = tmp->next;
```



```
prev->next = tmp->next;
```

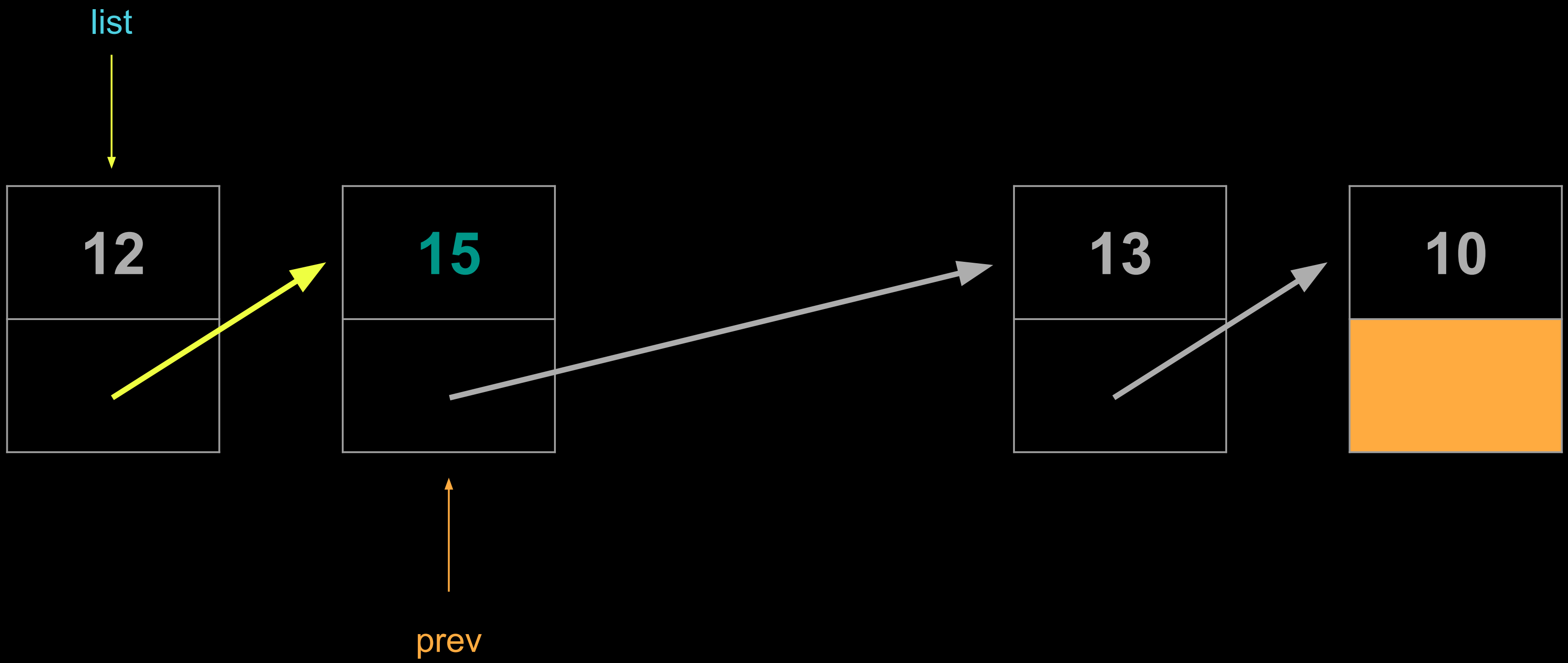


```
free(tmp);
```



Note: `->` operators can be chained!

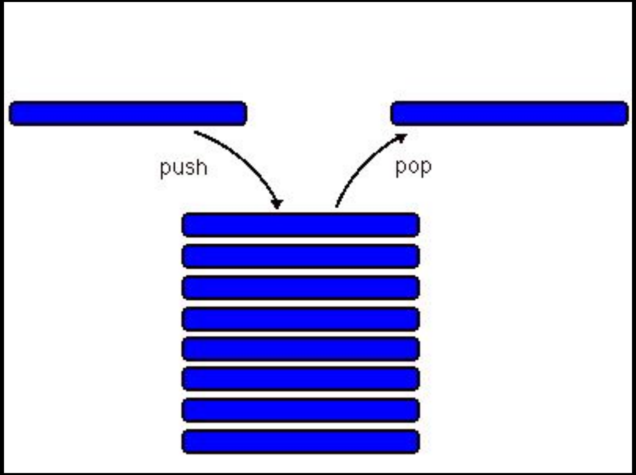
```
list->next->value
```



stacks and queues

Stacks

- Commonly an array or linked list
- Last-in-first-out (LIFO)
- 2 operations:
  - Push
  - Pop



Queues

- Also arrays or linked list, but have to keep track of “front” of the queue (previous and next pointers)
- First-in-first-out (FIFO)
- 2 Operations:
    - Enqueue
    - Dequeue

Implementation as linked list nodes

```
typedef struct stack
{
    int value;
    struct stack *next;
}
stack;
```

(pretty much a regular linked list!)

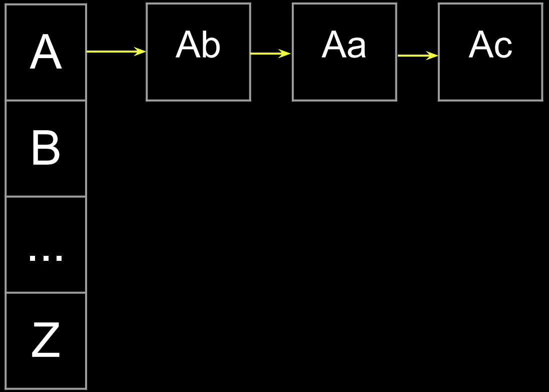
```
typedef struct node
{
   int value;
   struct node *next;
}
node;
typedef struct queue
{
    int count;
    node *front;
    node *rear;
}
queue;
```

Note: can be done in other ways, too!

Hash tables

An array of linked lists!

- Combines benefits of both
  - Random access to "bins"
  - Shorter linked lists to traverse
- Hash function determines which "bin" to use
  - Must be deterministic, only return a fixed number of choices



Hash functions

- A hash function finds the appropriate “bucket” for data
- A good hash function should:
    - Use **only** the data being hashed
    - Use **all** of the data being hashed
    - Generate the **same result** every time given same input; no randomness!
    - Uniformly distribute data
    - Generate very different hash codes for very similar data.

...ahem

```c
const int N = 1000000
// hash function described in http://www.cse.yorku.ca/~oz/hash.html
unsigned int hash(const char *str)
{
   unsigned int hash = 5381;
   char c;
   while ((c = *str++))
   {
       hash = ((hash << 5) + hash) + tolower(c);
   }
   return hash % N;
}
```

Note: use a **big** number for N!

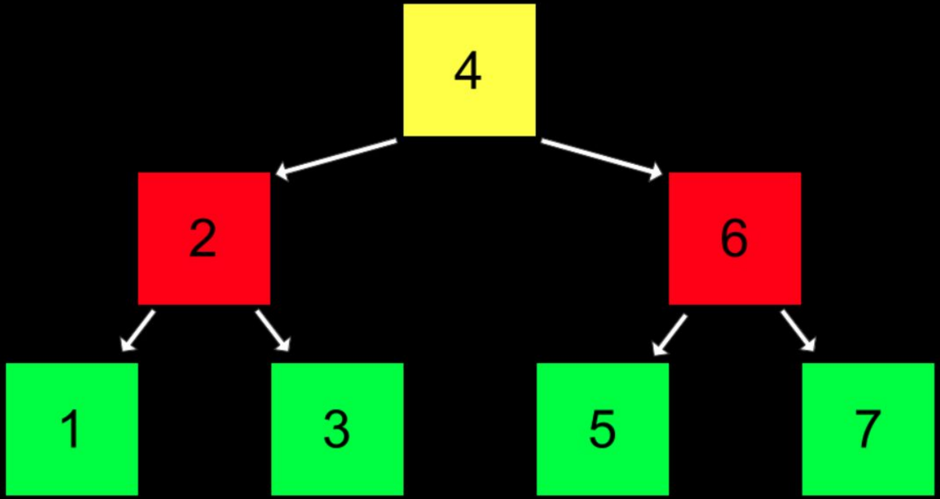
Collisions

- Unless your hash function is perfect, collisions will happen
- Solution 1: Linear probing (not ideal in our case)
    - If we have a collision, try to place the data in the next consecutive element of the array instead
    - Try each consecutive element until we find a vacancy.
    - That way, if we don’t find it right where we expect it, it’s hopefully nearby.
- Solution 2: Chaining (with linked lists)
    - Each element of the array is now a pointer to the head of a linked list.
    - For a collision, make a new node and add it to the chain at that location.
    - The item is where it’s expected, just in a list

trees

Binary search tree

- Similar to a linked list, but instead of `next`, it has 2 pointers: `left` and `right`
- To be effective:
  - Data must be sorted
  - The tree must be balanced (~same number of entries on each side)
- Gives the speed of binary search with the flexibility of a linked list



tries

Tries

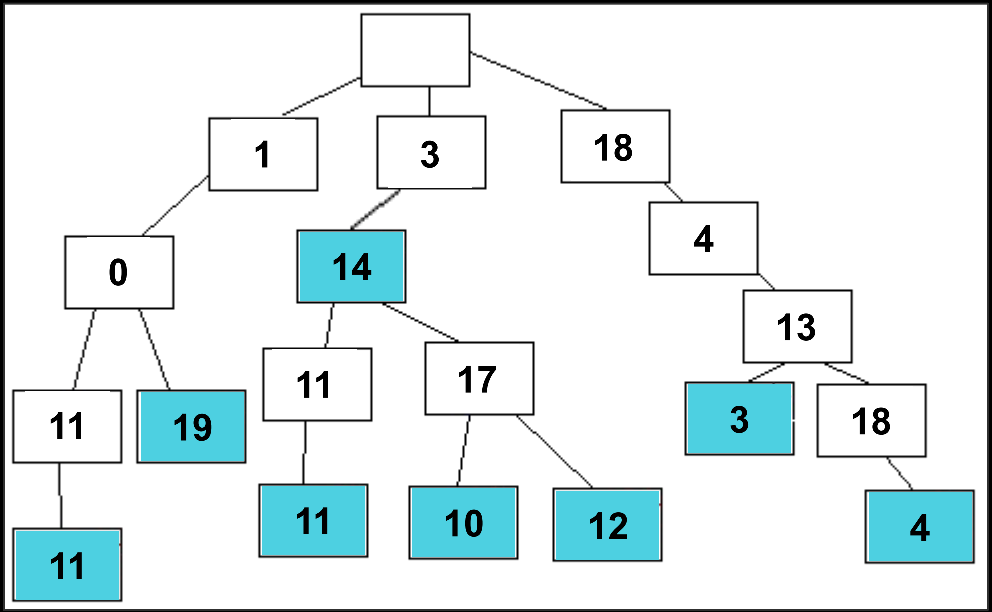
- Tries have unique keys, with values as simple as a `bool` to tell you that the data exists in the structure
- No collisions – no two pieces of data have the same path
- Insertion, deletion, and lookup all O(1)!
  - In reality, takes up to *n* steps, where *n* is the height of the trie
- Tradeoff is space!

Trie `struct` for dictionary use

```
typedef struct node
{
   bool is_word; // 1 byte
   struct node *children[27] // 8 bytes * 27 = 216
}

node;
```

- 27 needed for 26 letters + ' (apostrophe)
- Word is never “stored” as a whole -- only the path

Graphical trie



Questions?