CSE 4202: Structured Programming II Lab

Lecture 6 — Linked Lists

Syed Rifat Raiyan
Lecturer
Department of Computer Science & Engineering
Islamic University of Technology, Dhaka, Bangladesh
Email: rifatraiyan@iut-dhaka.edu

Lecture Plan

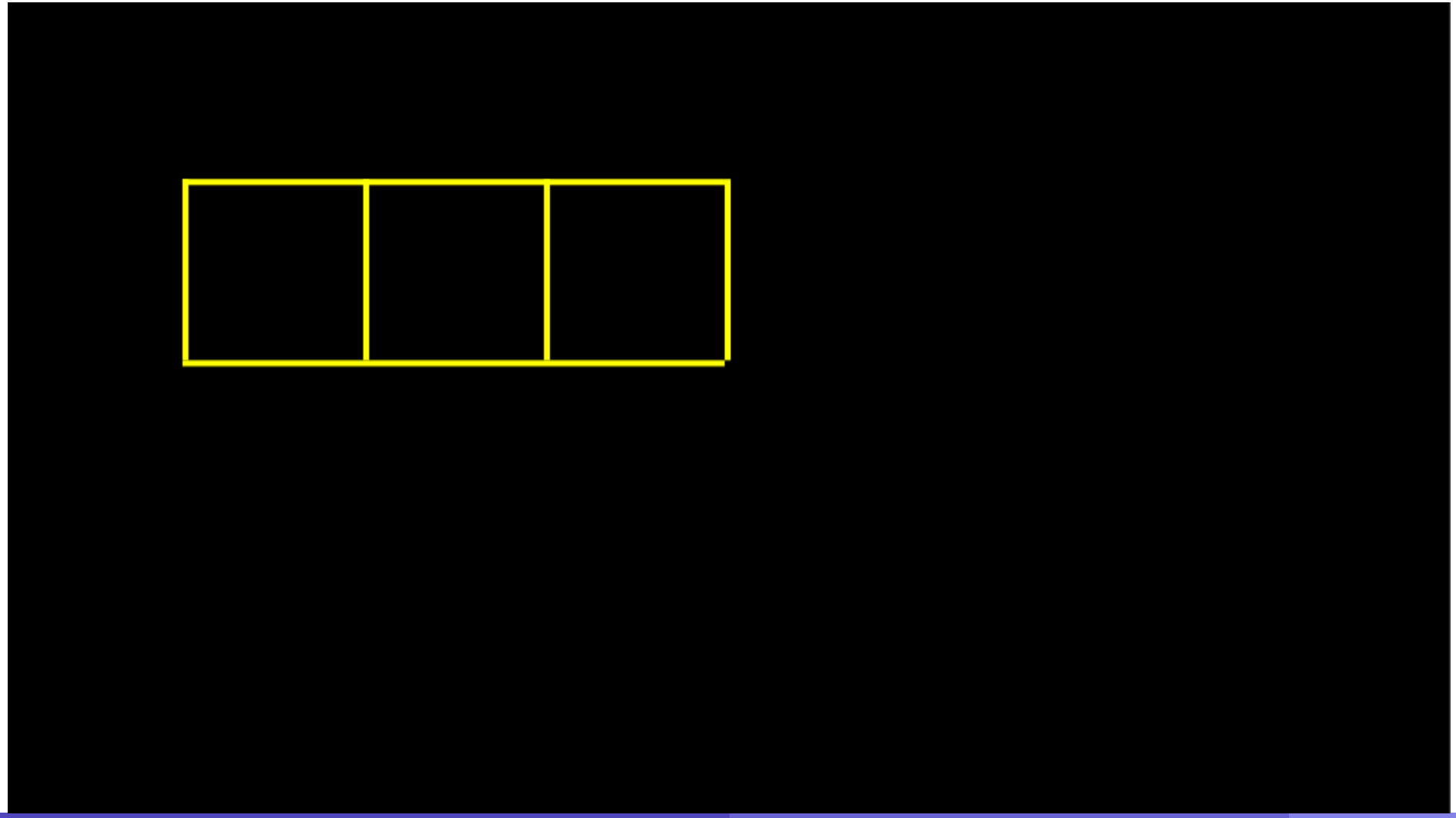
The agenda for today

- Recap on resizing arrays dynamically
- Pictorial overview of linking
- Creating a single node
- Different ways of inserting a single node
- Printing the list via traversal
- Searching for a value
- Deleting the entire linked list (recursively)
- Different ways of deleting a single node

Dynamically Resizing Arrays

What we did so far…

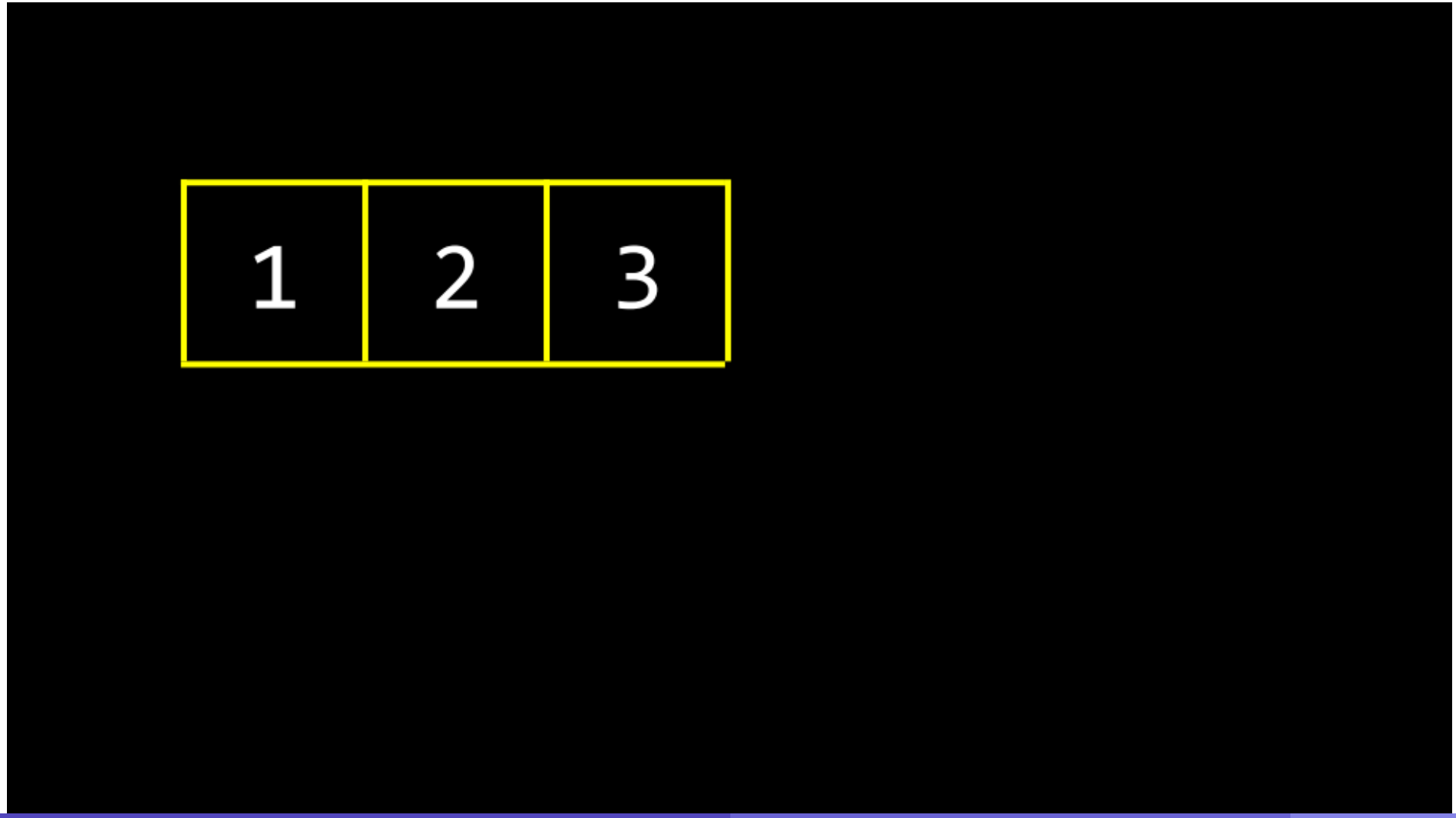
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

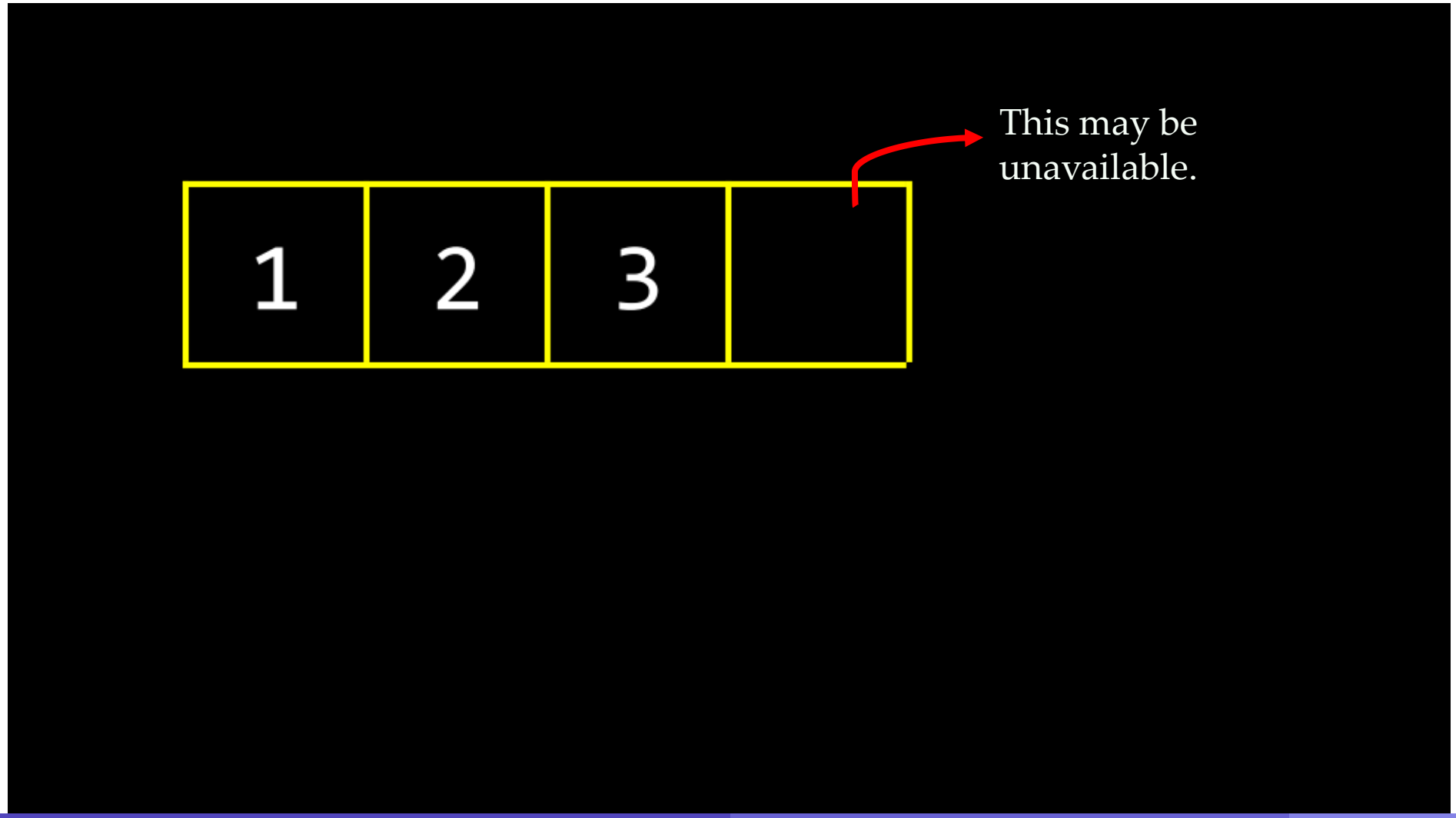
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

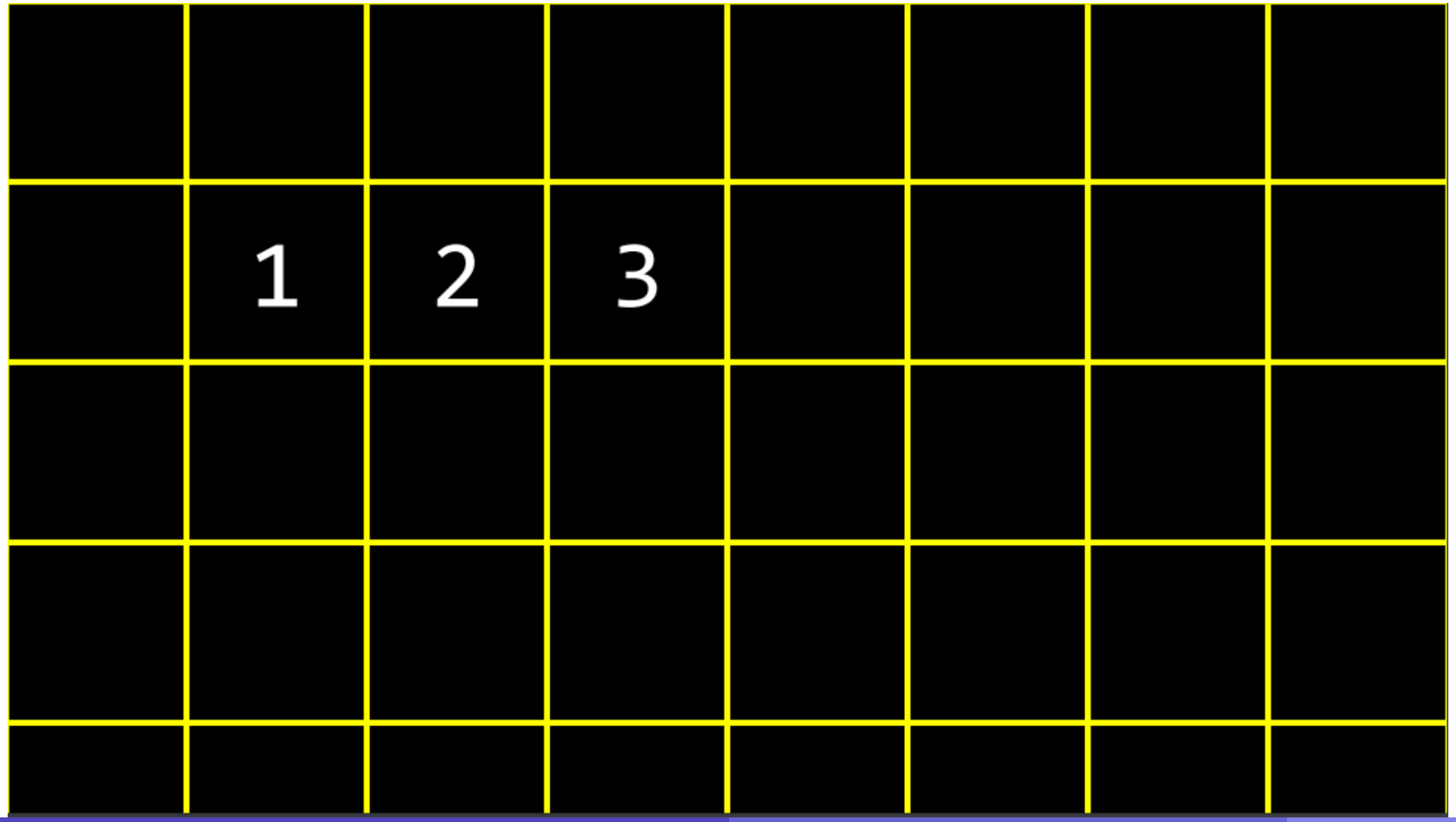
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

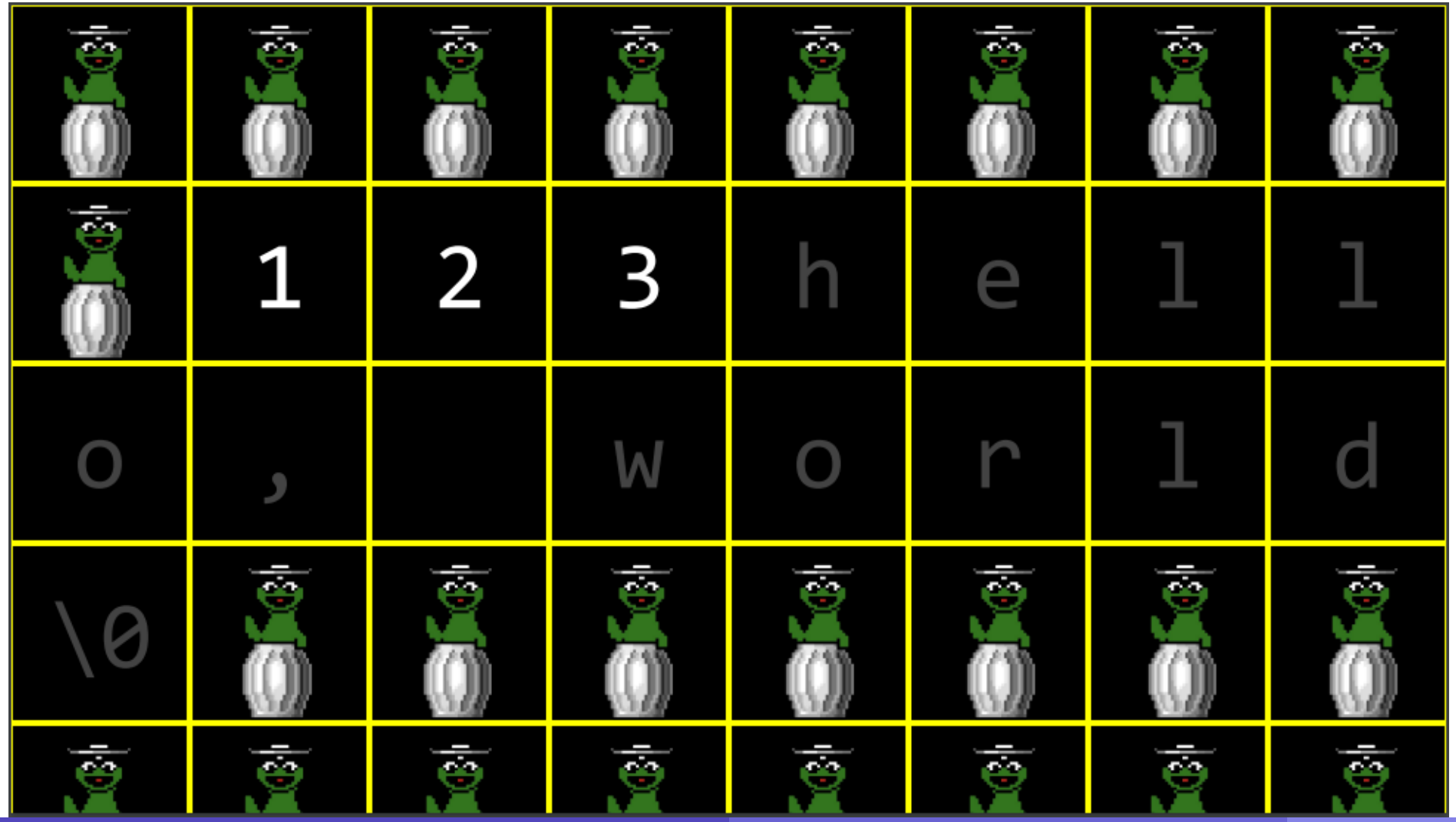
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

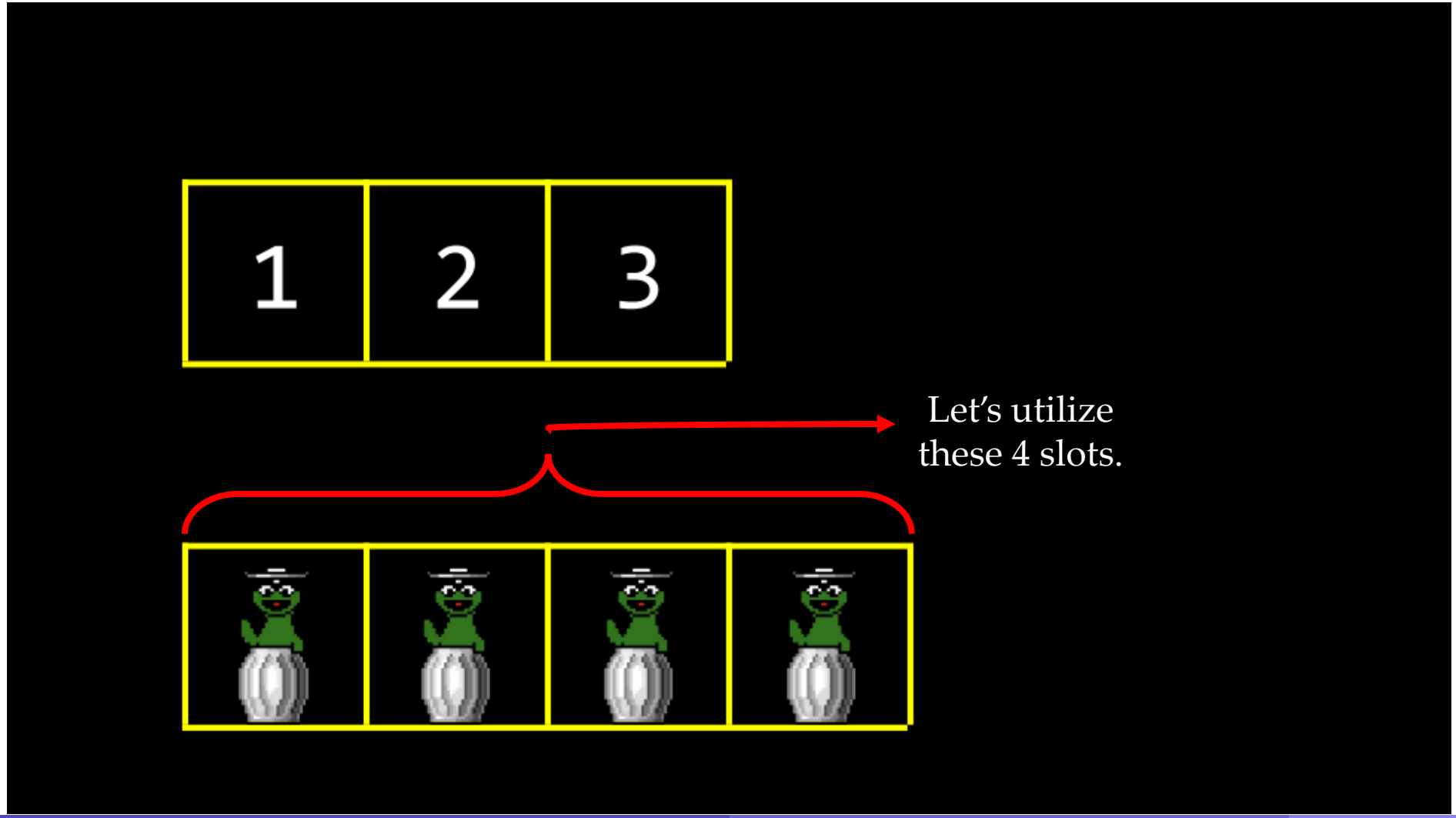
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

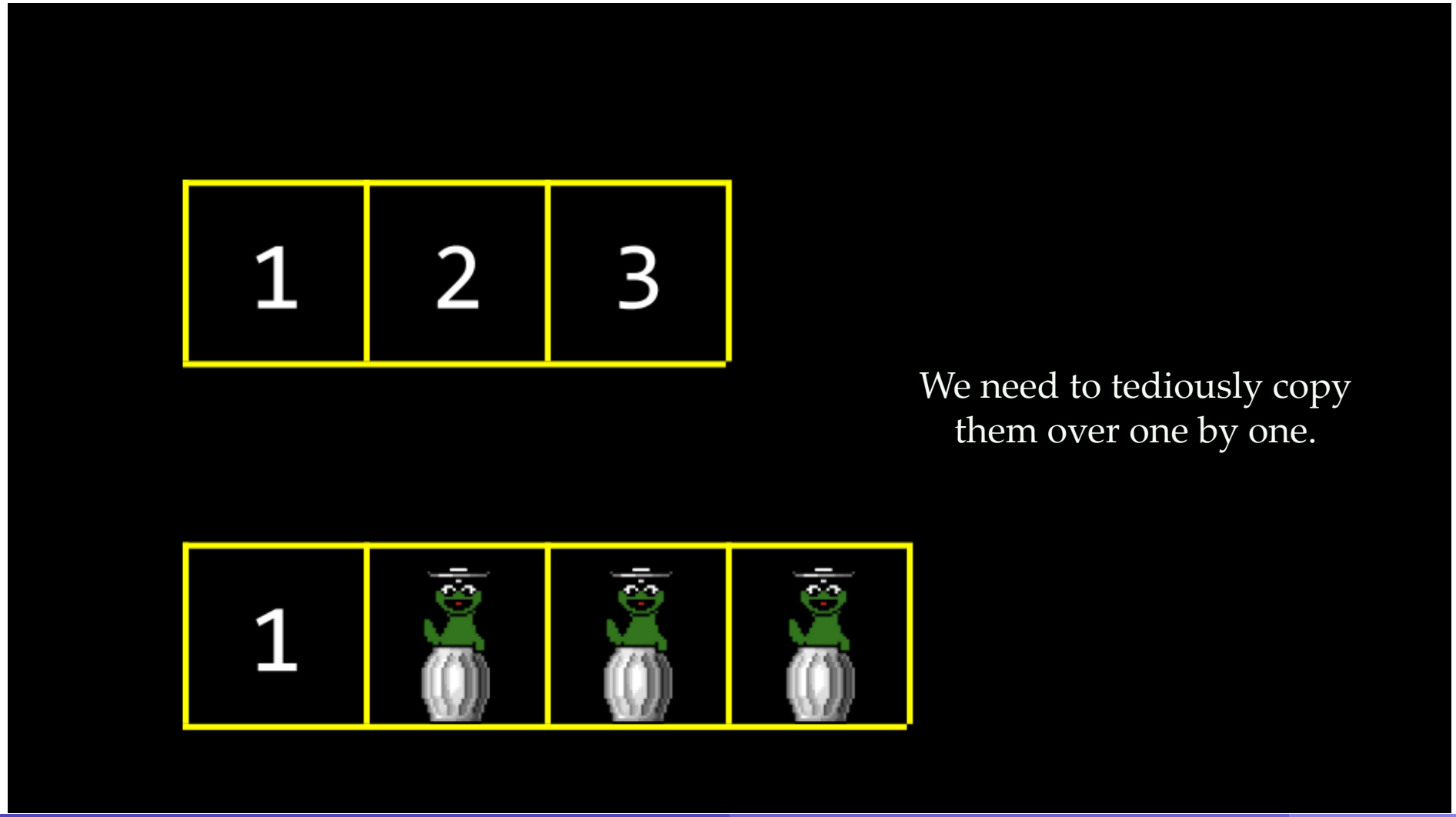
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

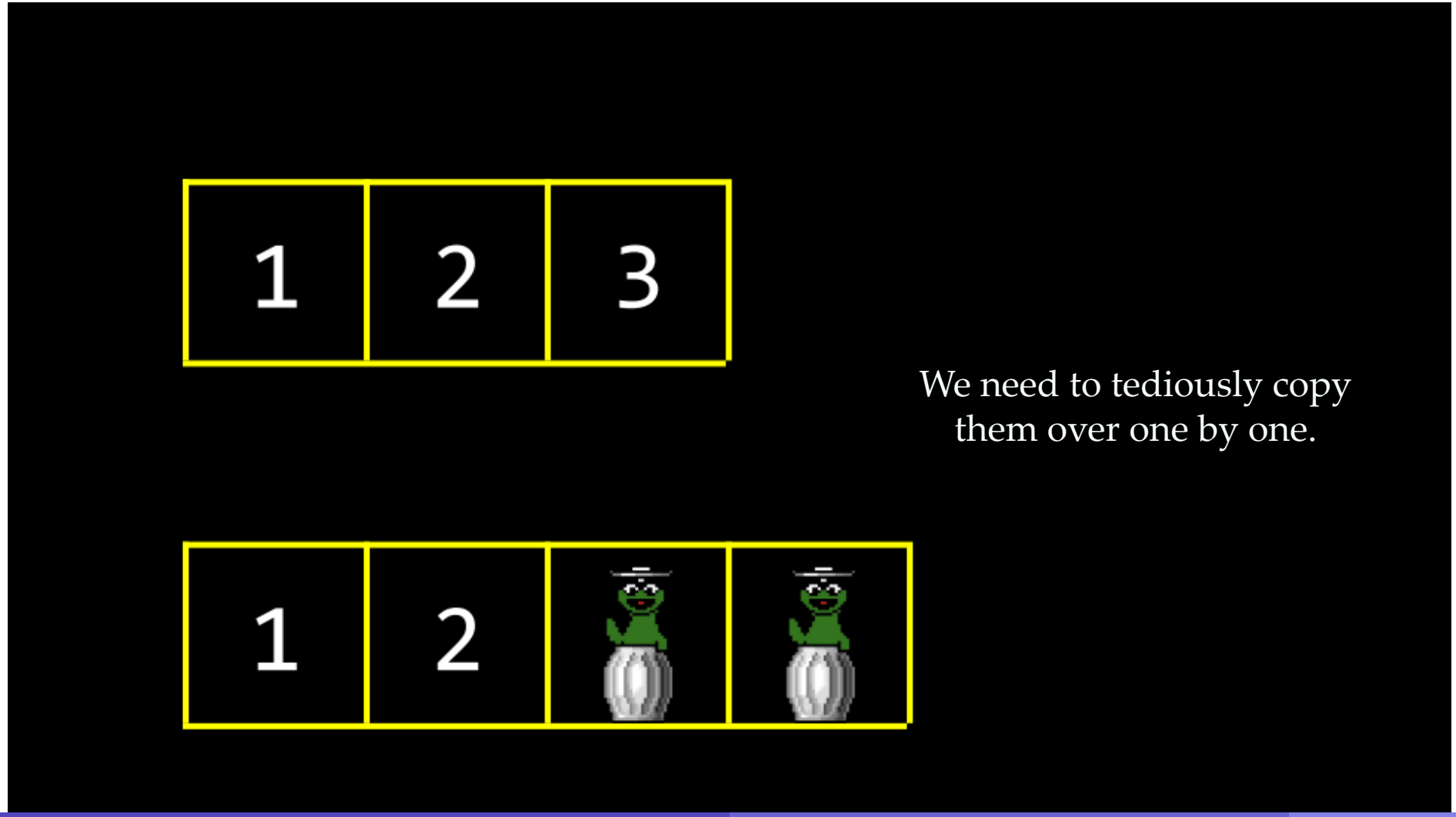
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

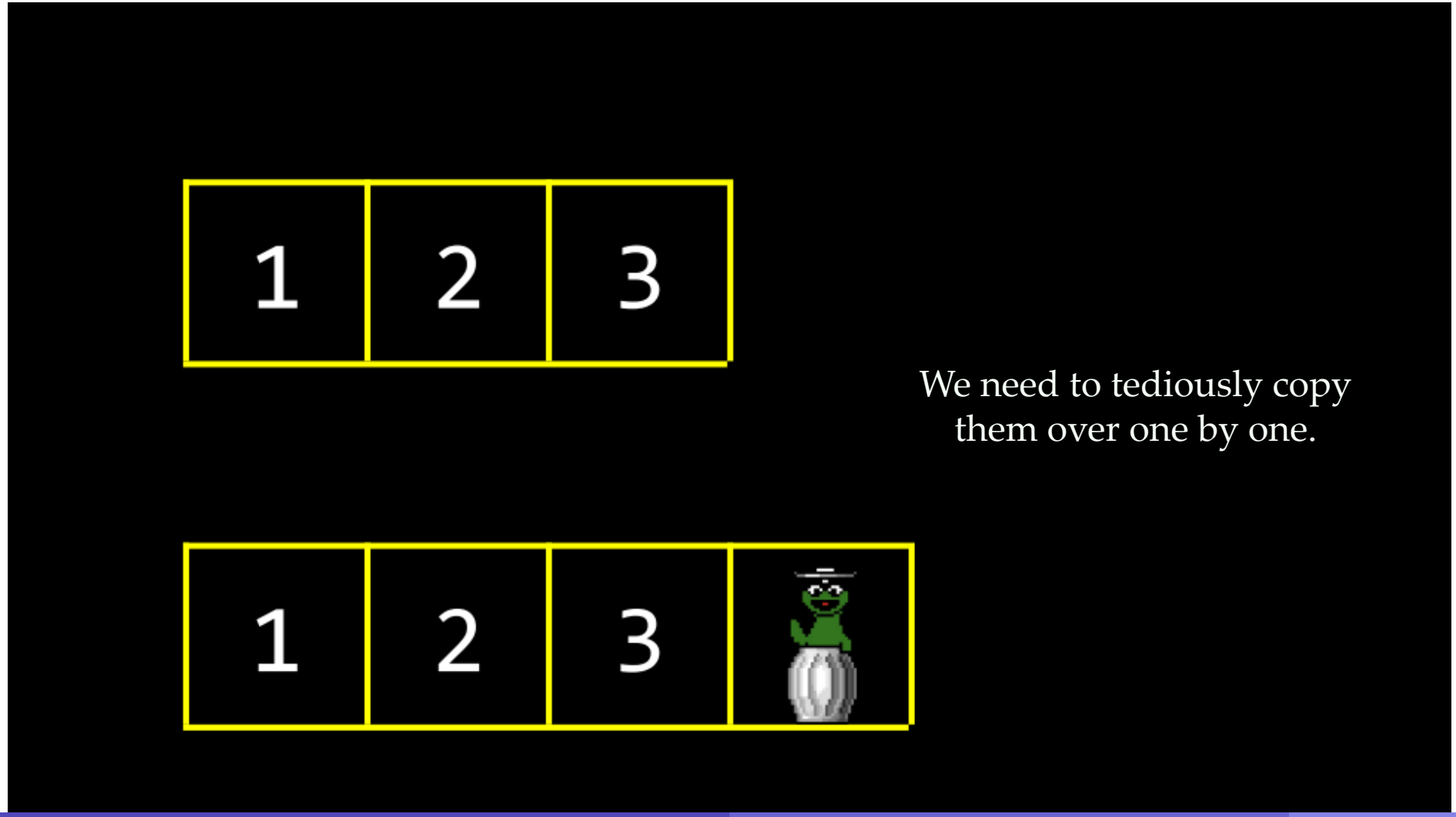
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

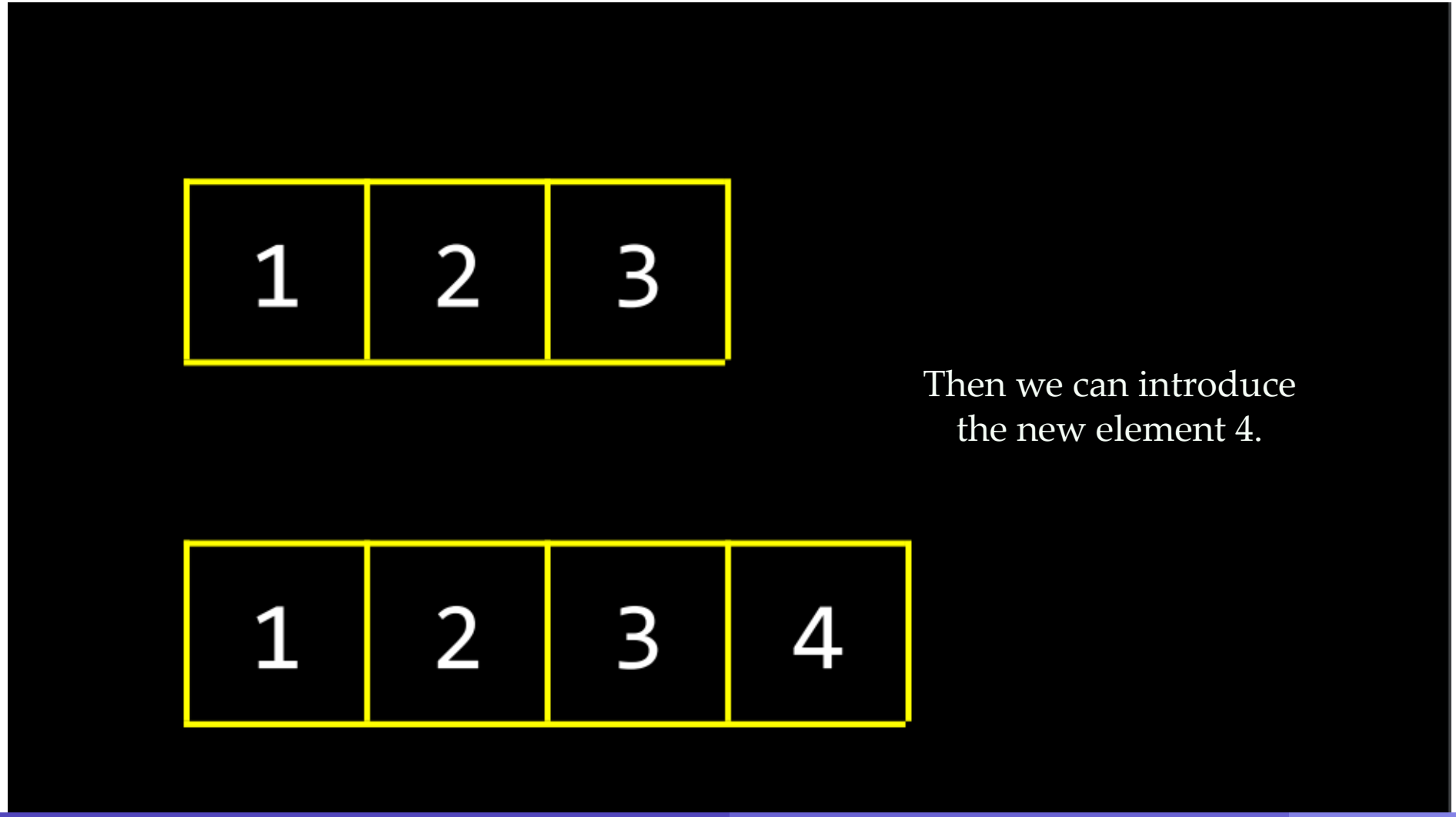
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

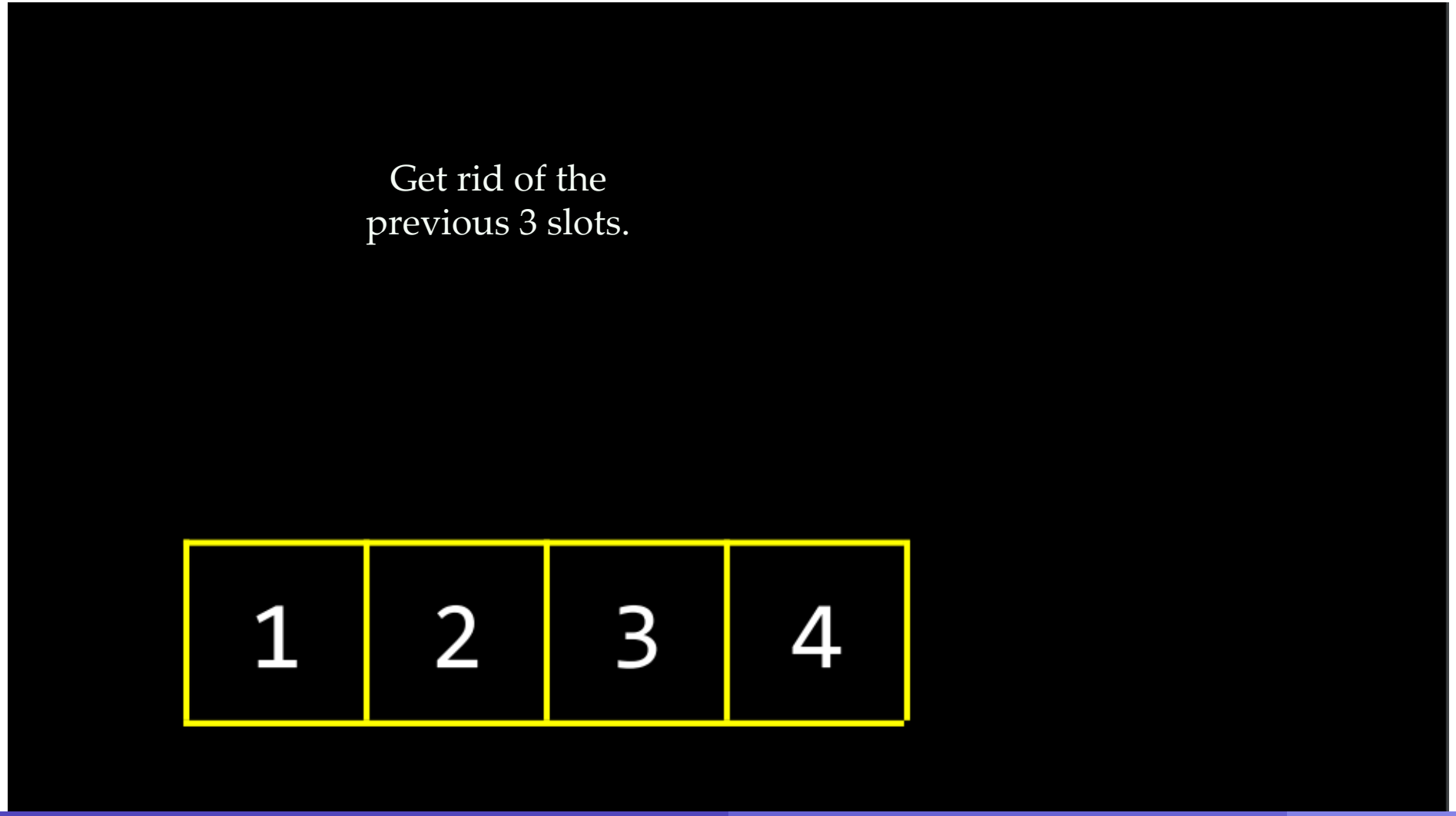
Upon requiring more memory, we resorted to **`realloc()`**.



Dynamically Resizing Arrays

What we did so far…

Upon requiring more memory, we resorted to **`realloc()`**.



Linked Lists

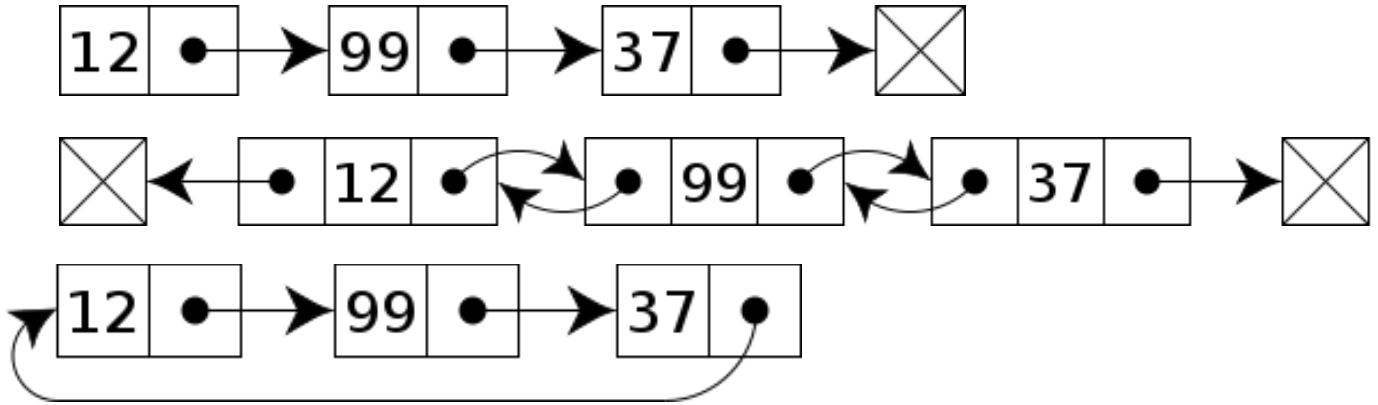
An overview

It is an **Abstract Data Type (ADT)** that allows us to utilize *non-contiguously located* memory slots as a linear sequence of elements by stitching them together.

- ✓ Other languages like Java, Python, and C++ readily offers us the **`list`** data structure.

Types of linked lists —

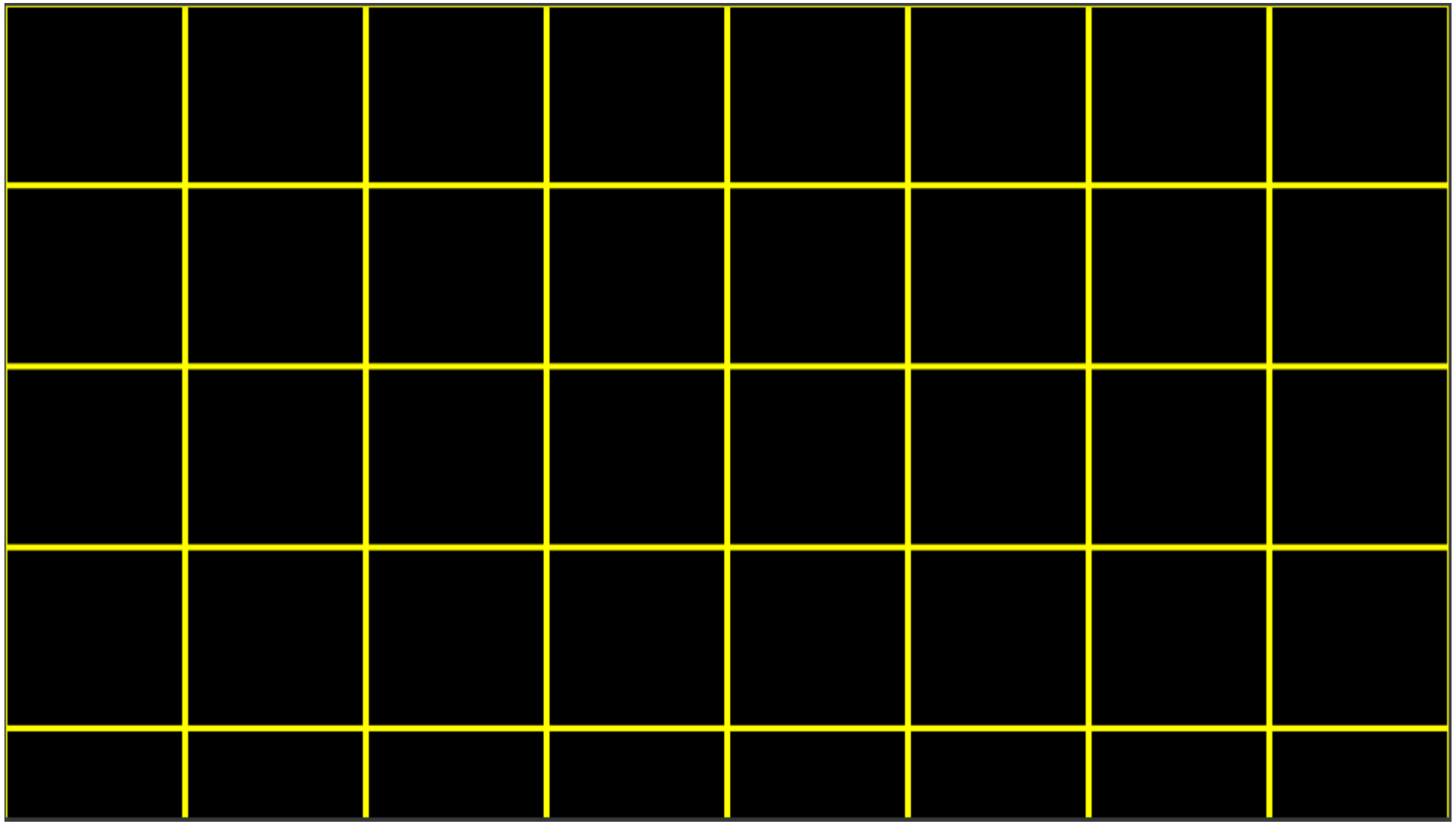
- Singly Linked List
- Doubly Linked List
- Circular Linked List
- and many more…



Prerequisite syntax: `struct`, `*`, `.`, `->`

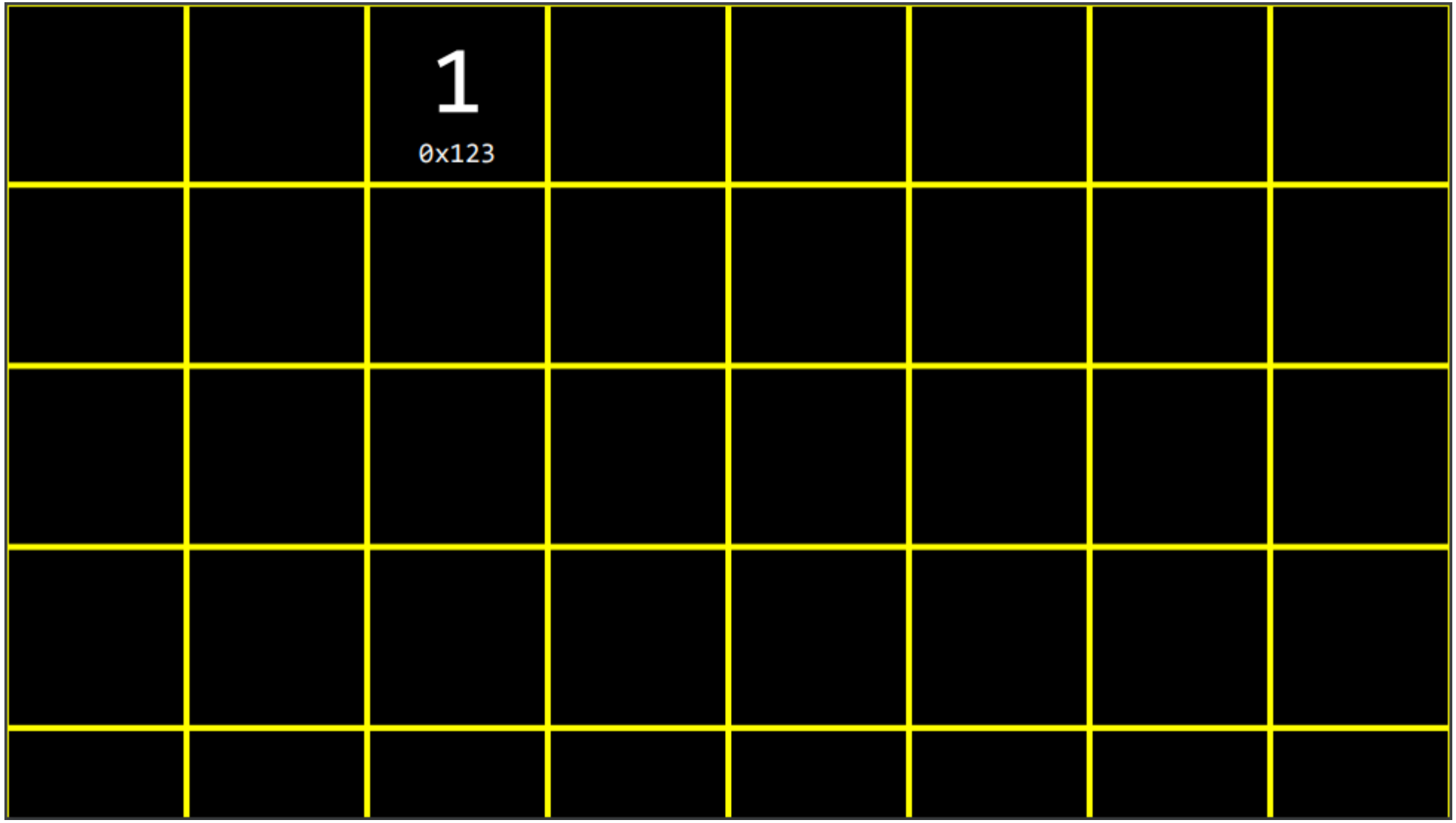
Linking

A visual walkthrough



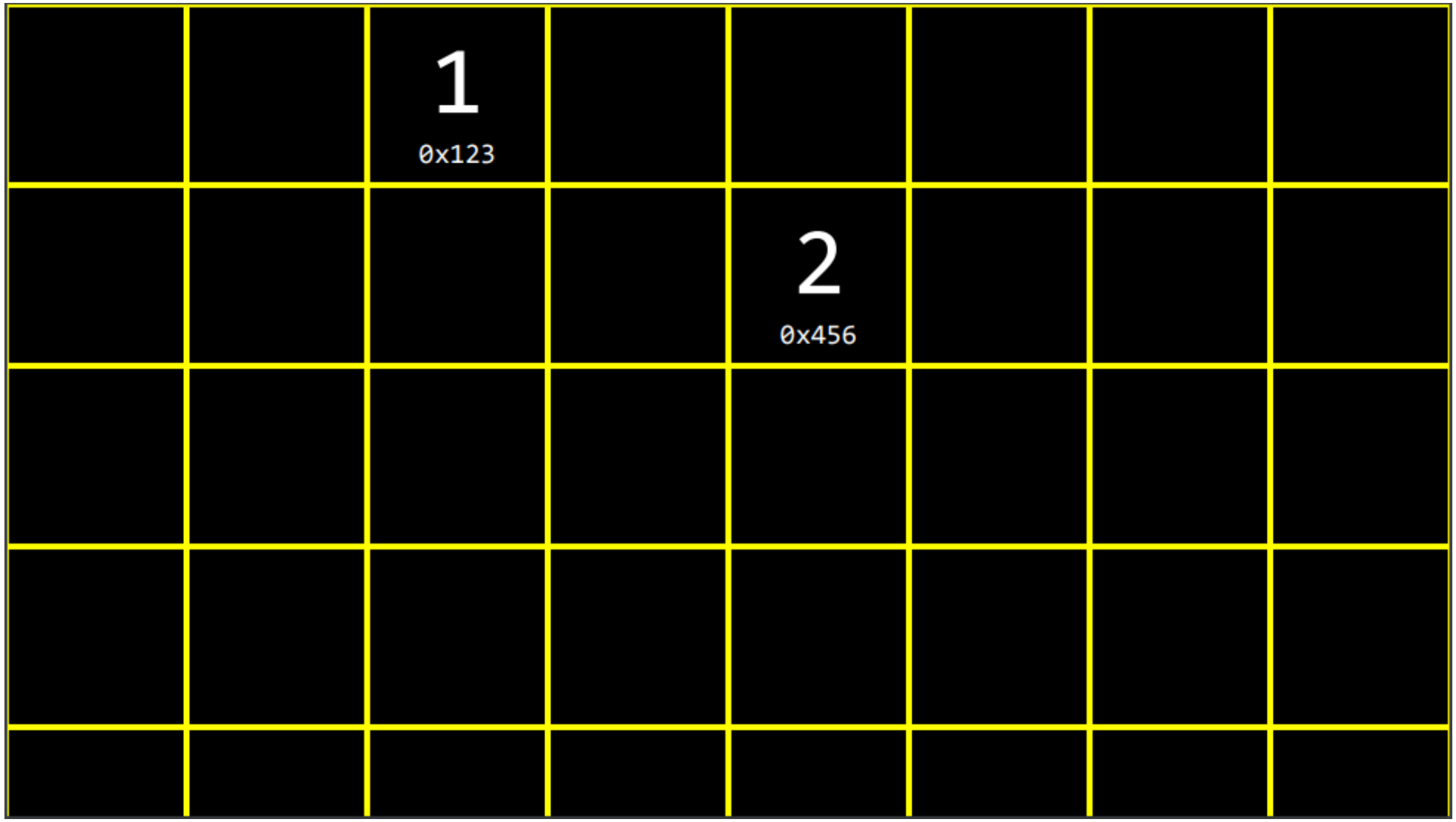
Linking

A visual walkthrough



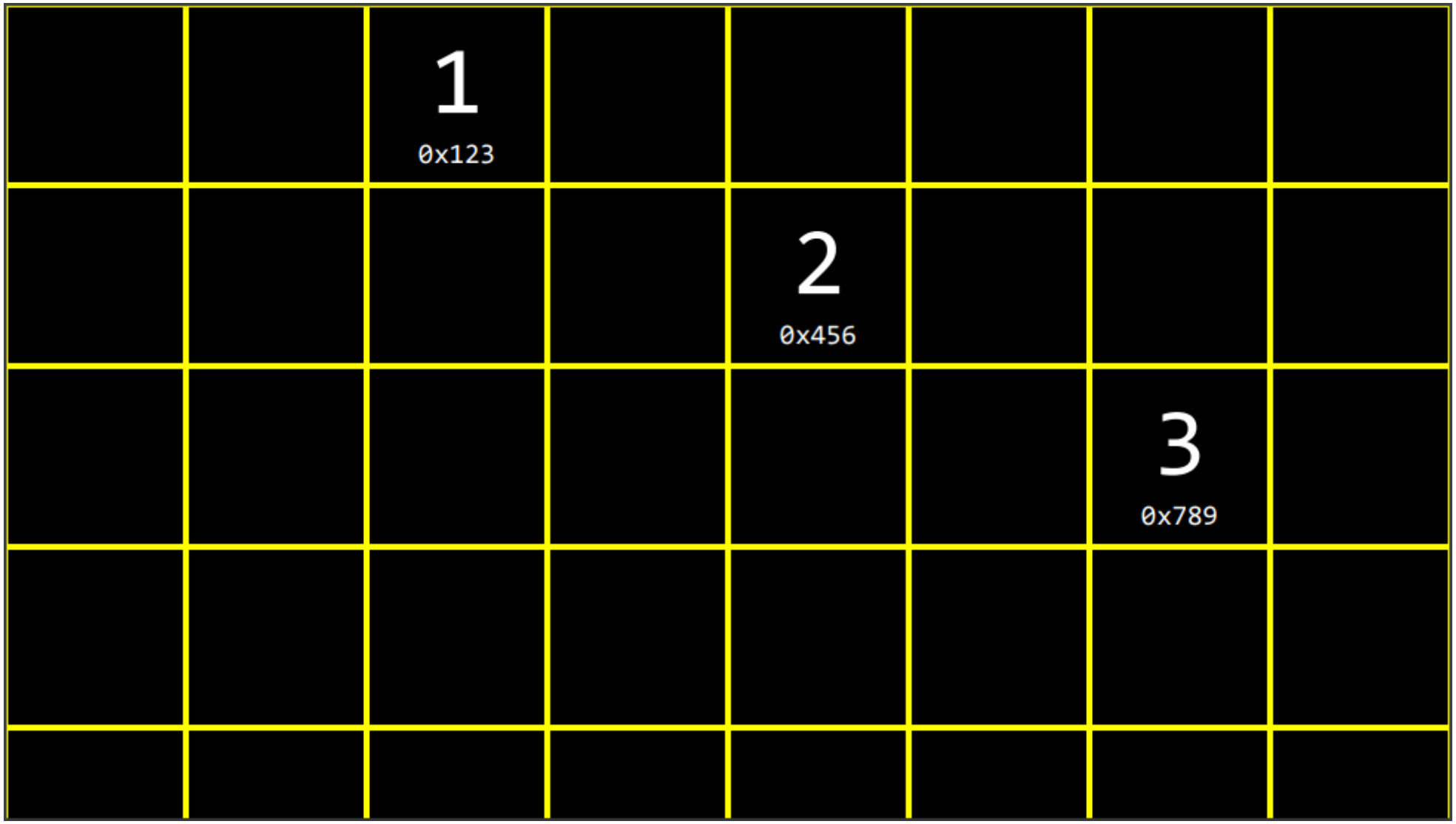
Linking

A visual walkthrough



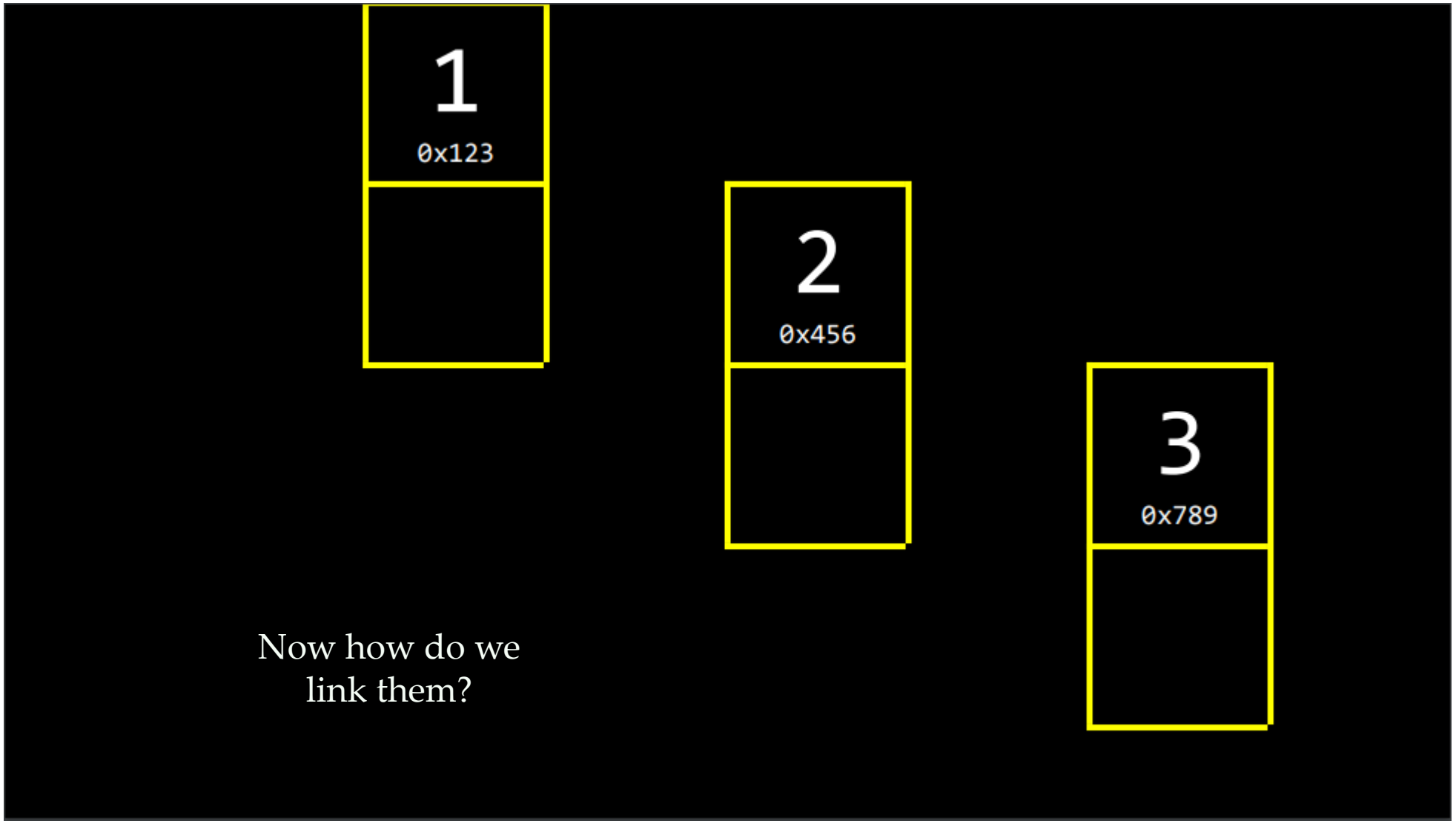
Linking

A visual walkthrough



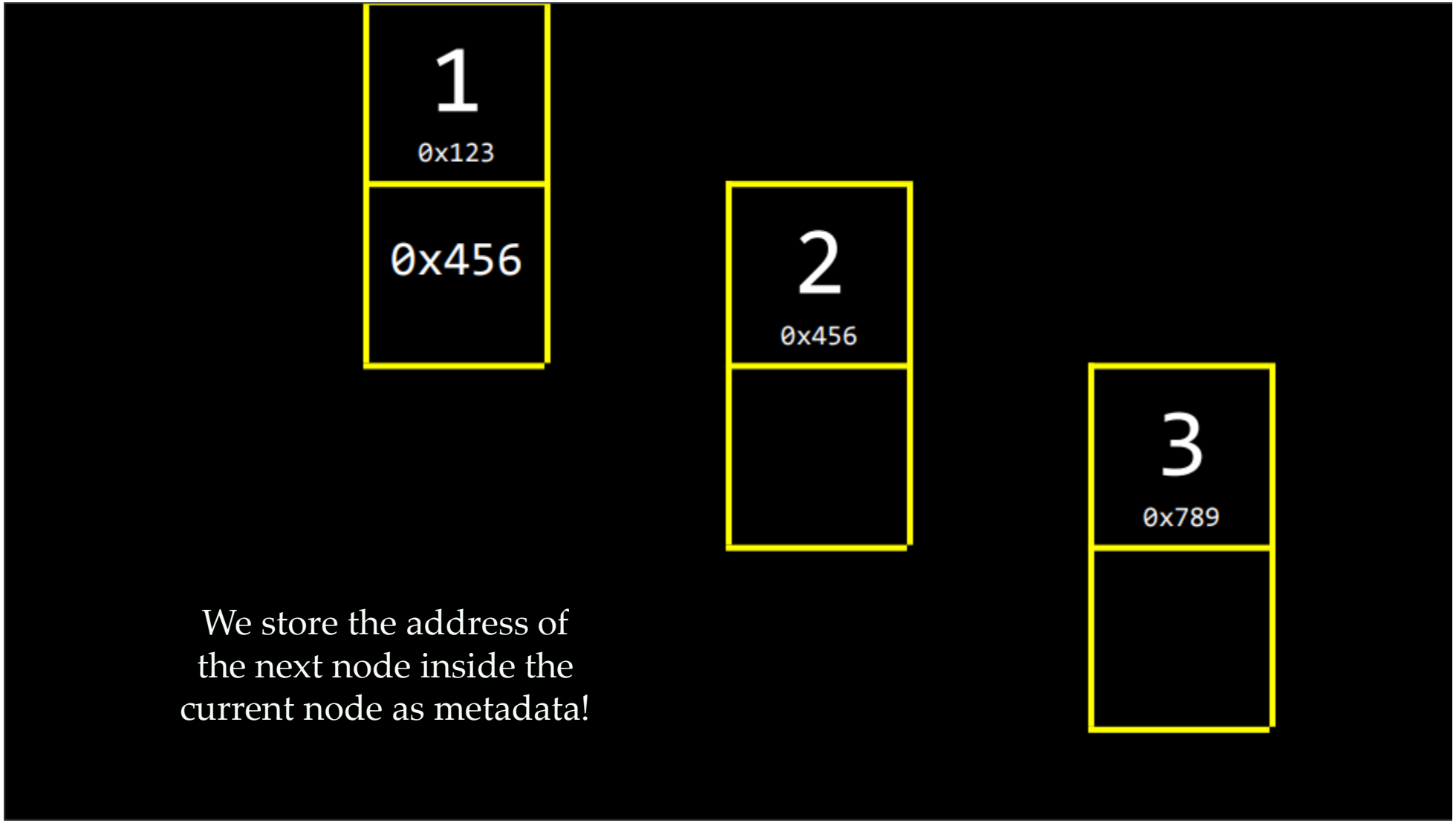
Linking

A visual walkthrough



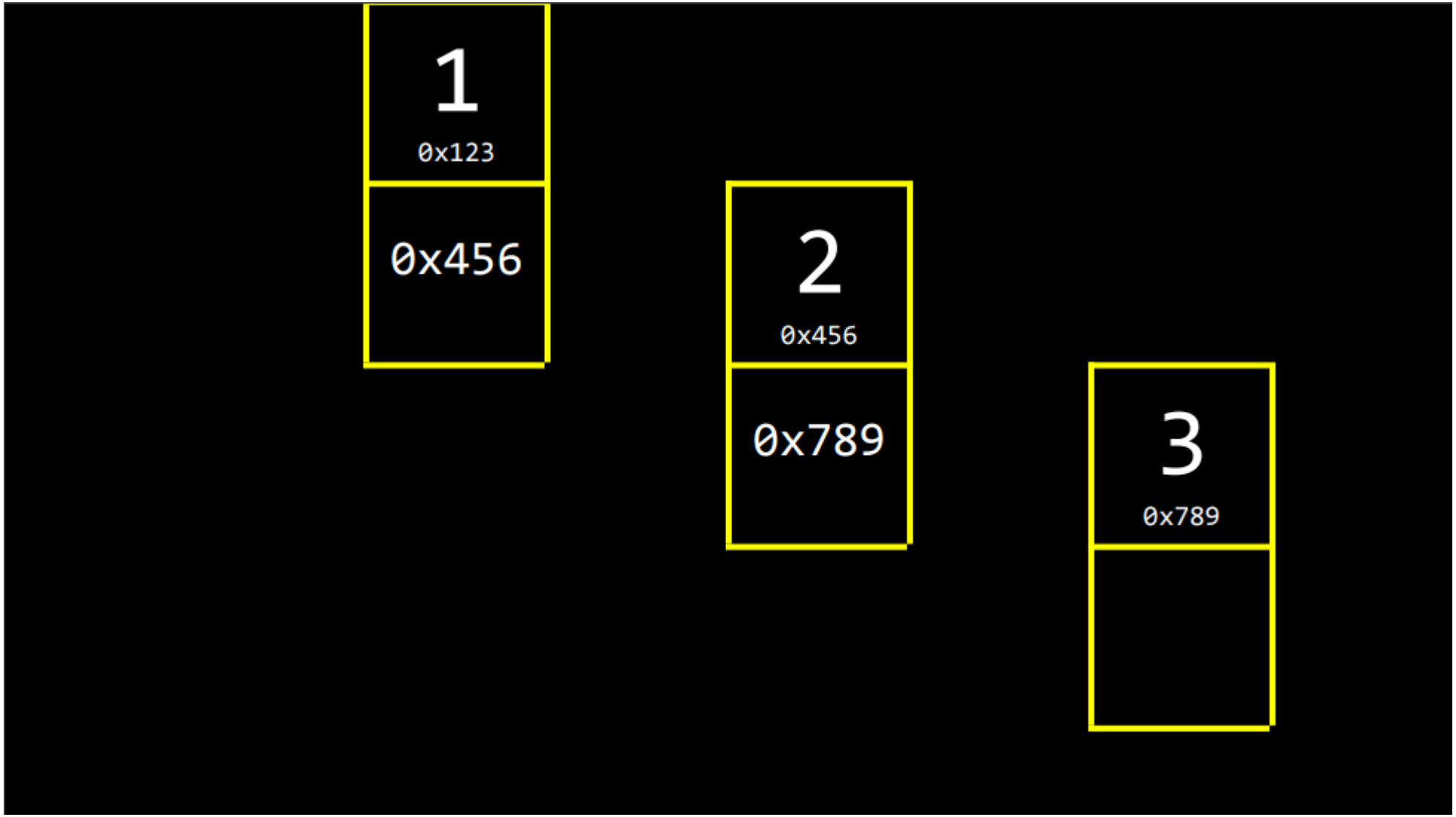
Linking

A visual walkthrough



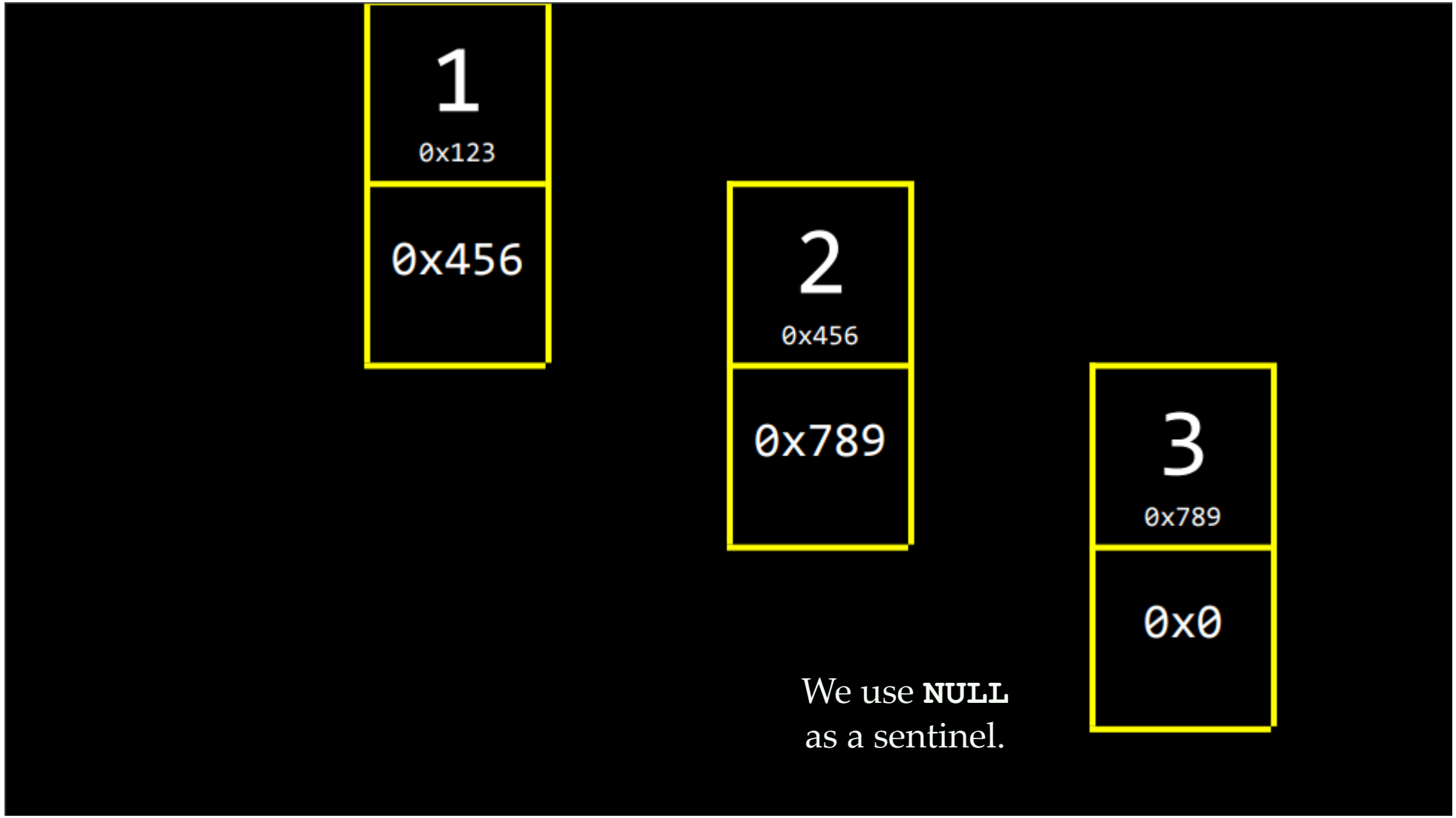
Linking

A visual walkthrough



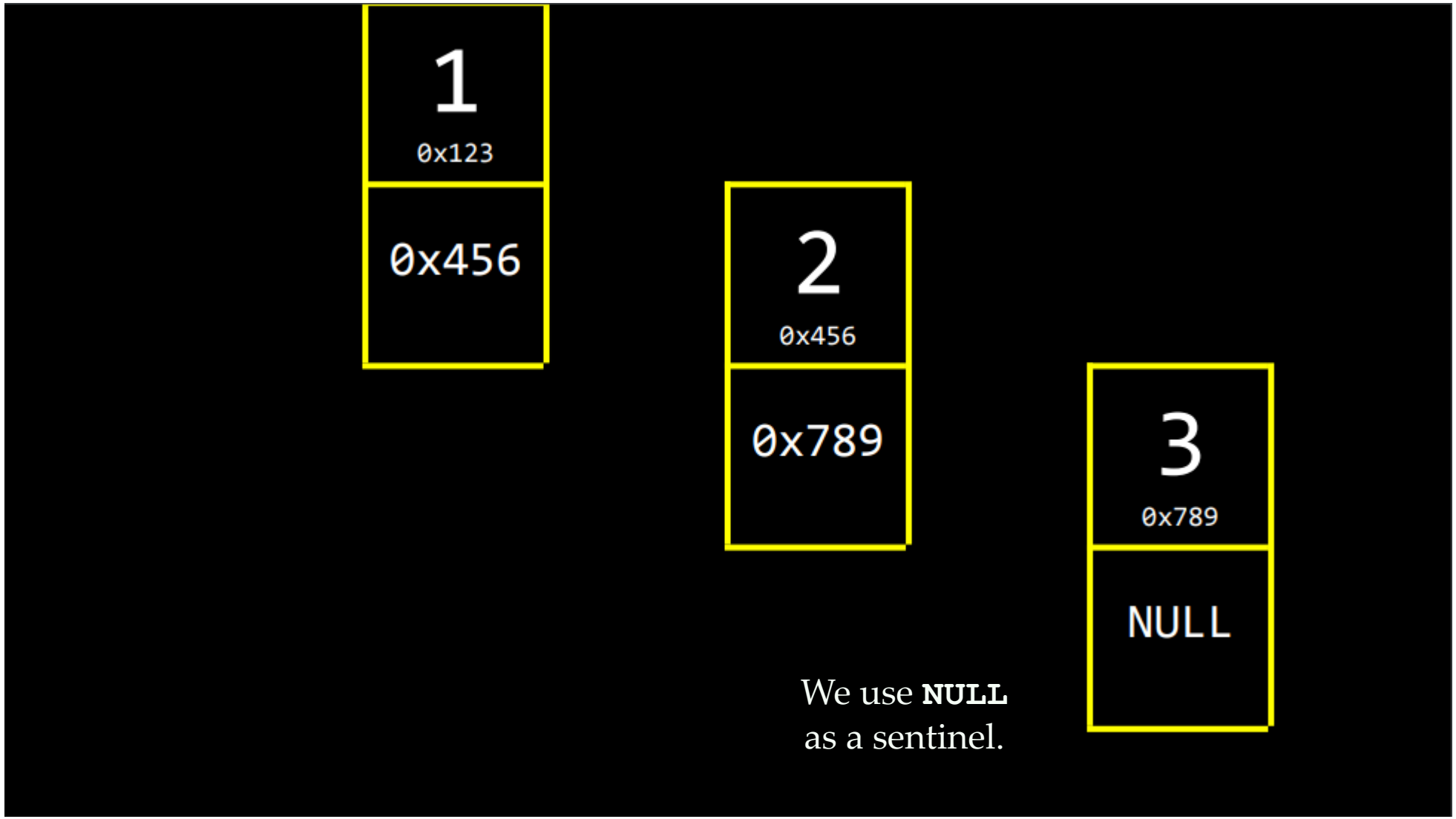
Linking

A visual walkthrough



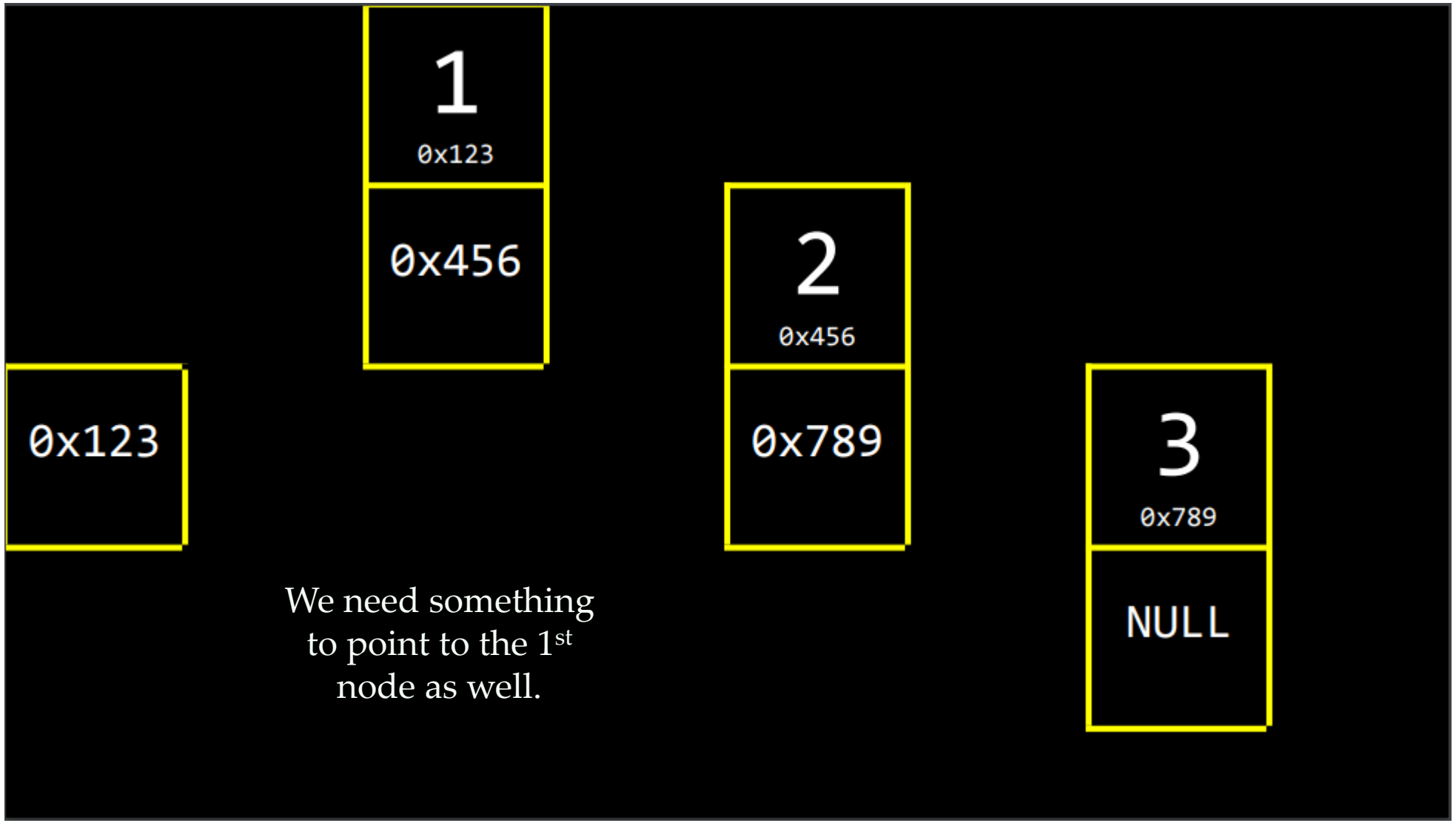
Linking

A visual walkthrough



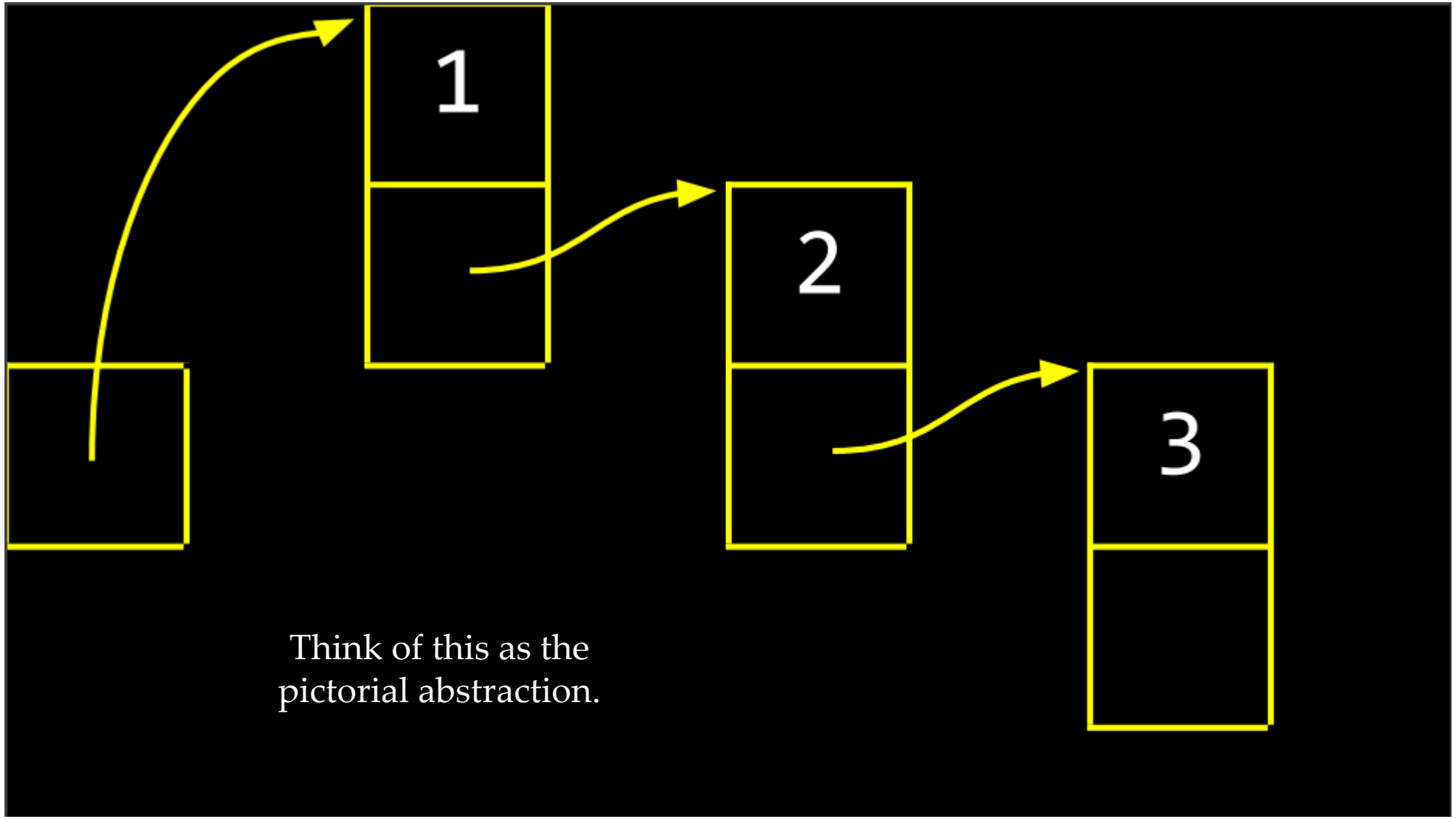
Linking

A visual walkthrough



Linking

A visual walkthrough



Singly Linked List

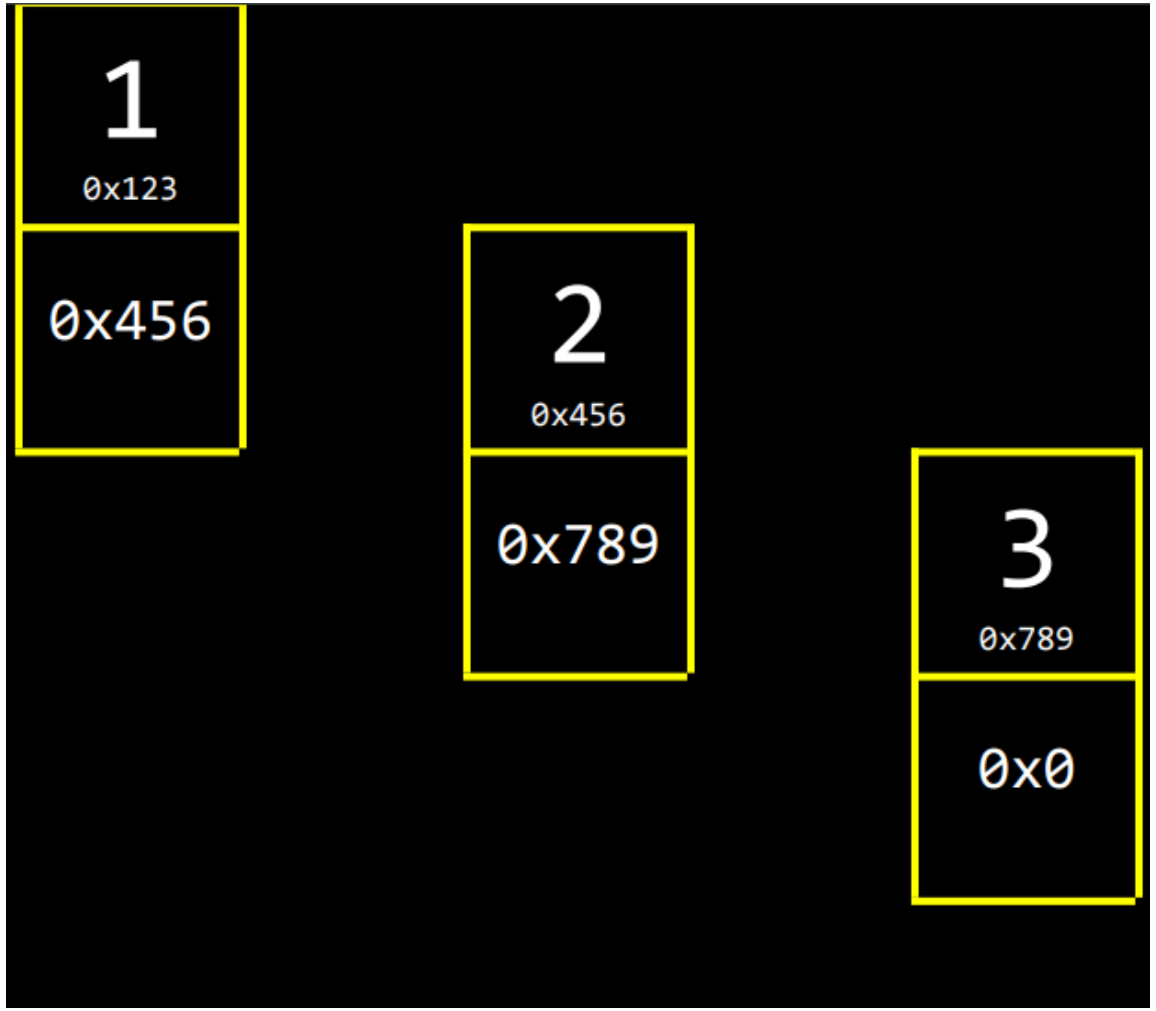
Node representation

Recall that, **`struct`**s give us *"containers"* for holding variables of different data types, typically.

A linked list node is a special kind of **`struct`** with two members:

- Data of some data type (**`int`**, **`char`**, **`float`**…)
- A pointer to another node of the same type

In this way, a set of nodes together can be thought of as forming a chain of elements that we can follow from beginning to end.

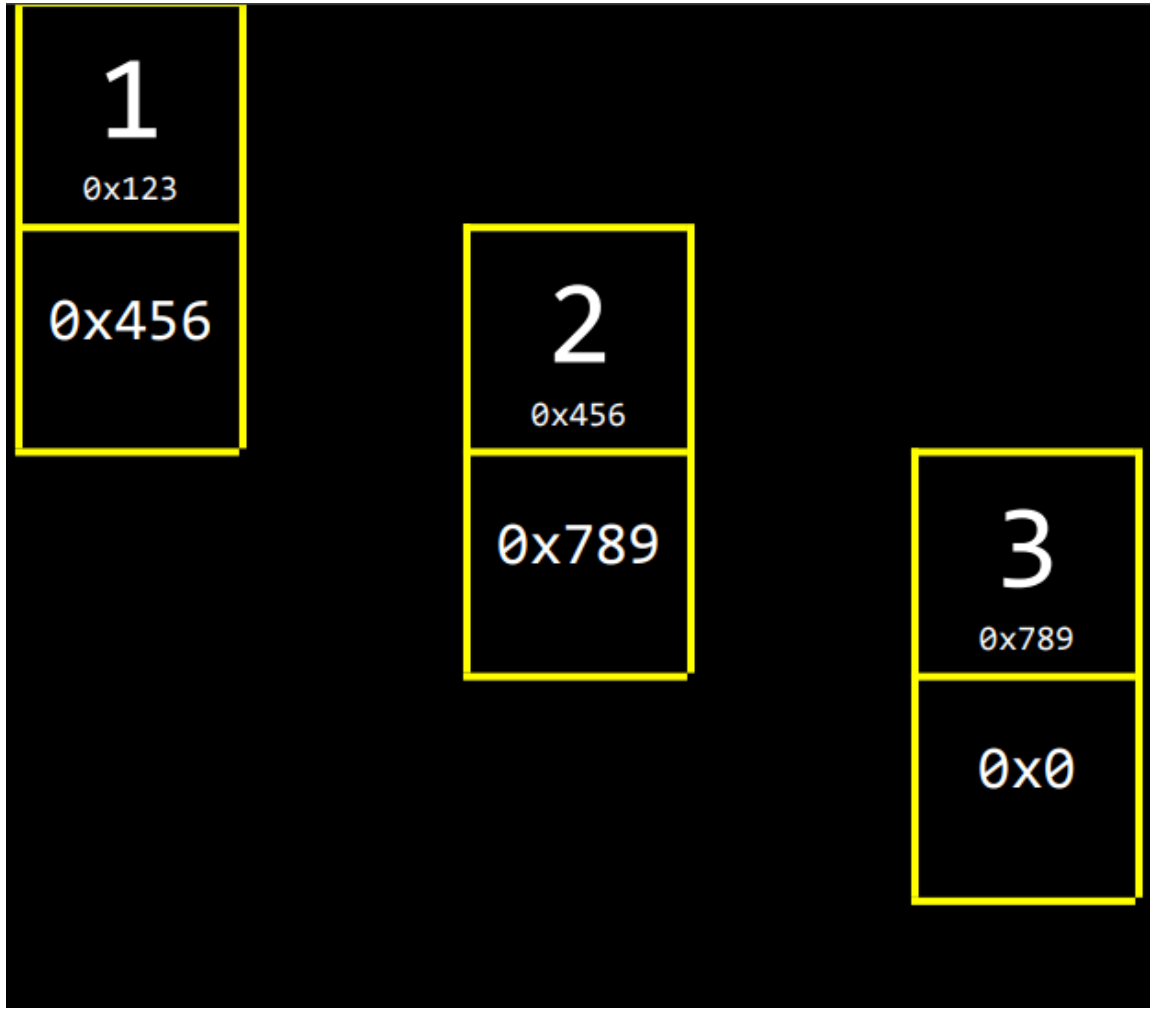


Singly Linked List

Node representation

Let's define the structure of the node.

```
typedef struct
{


} node;
```

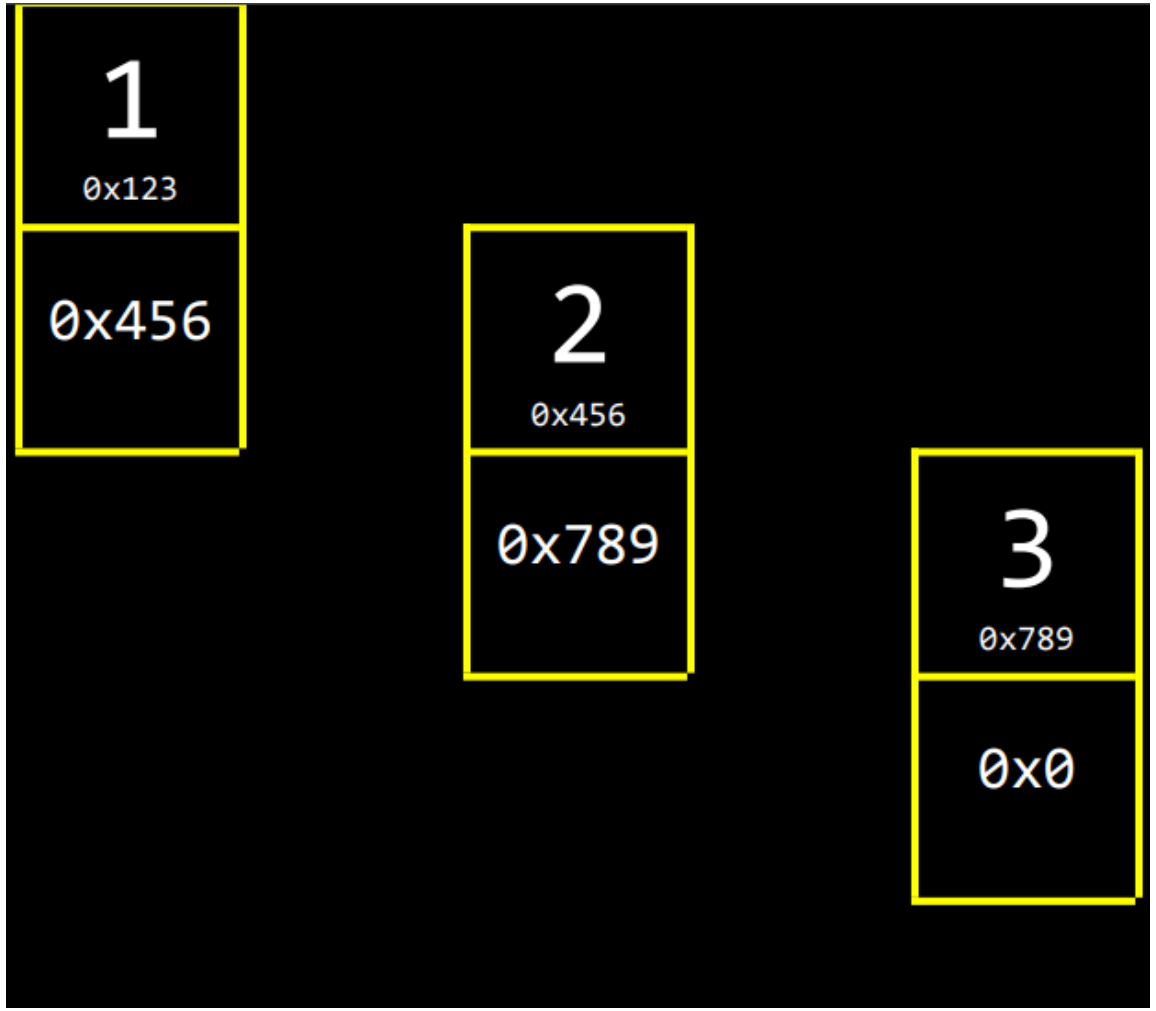


Singly Linked List

Node representation

We need a **`value`** field to store the number being stored within the node. What else do we need?

```
typedef struct
{
    int value;

} node;
```

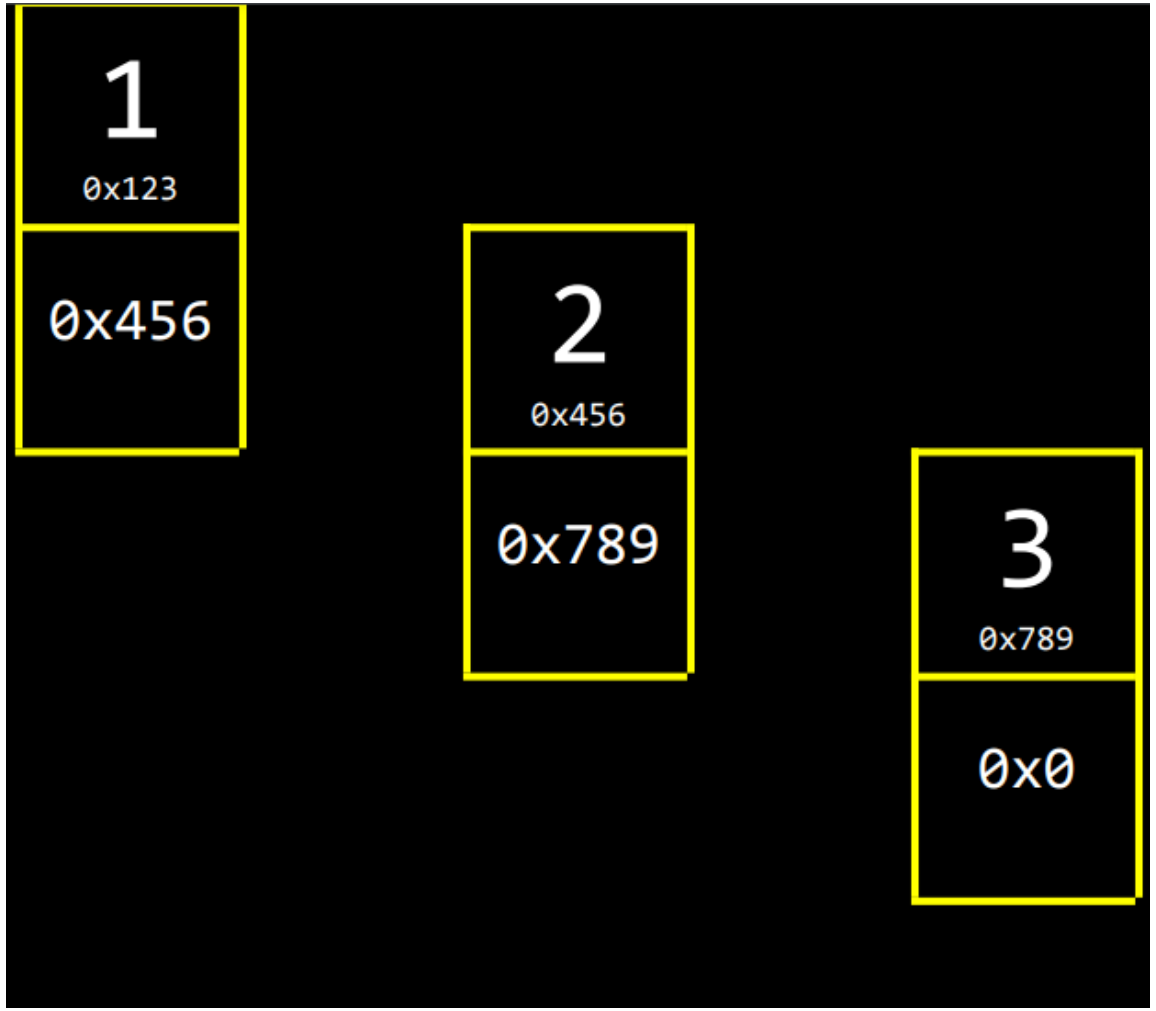


Singly Linked List

Node representation

We also need a **`next`** pointer to point to the next node in the list. But will this work?

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

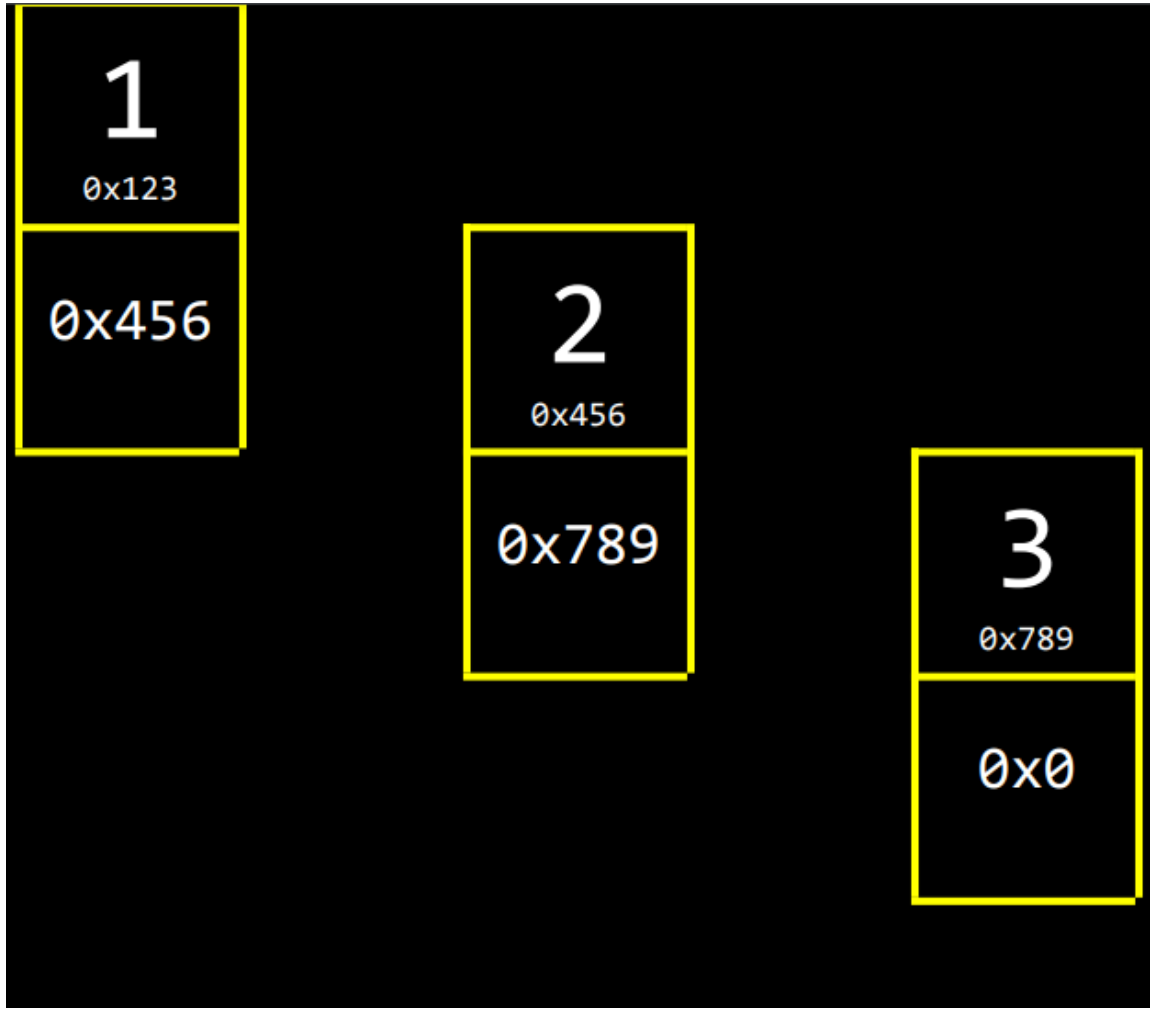


Singly Linked List

Node representation

Recall that C parses from top to bottom and from left to right.

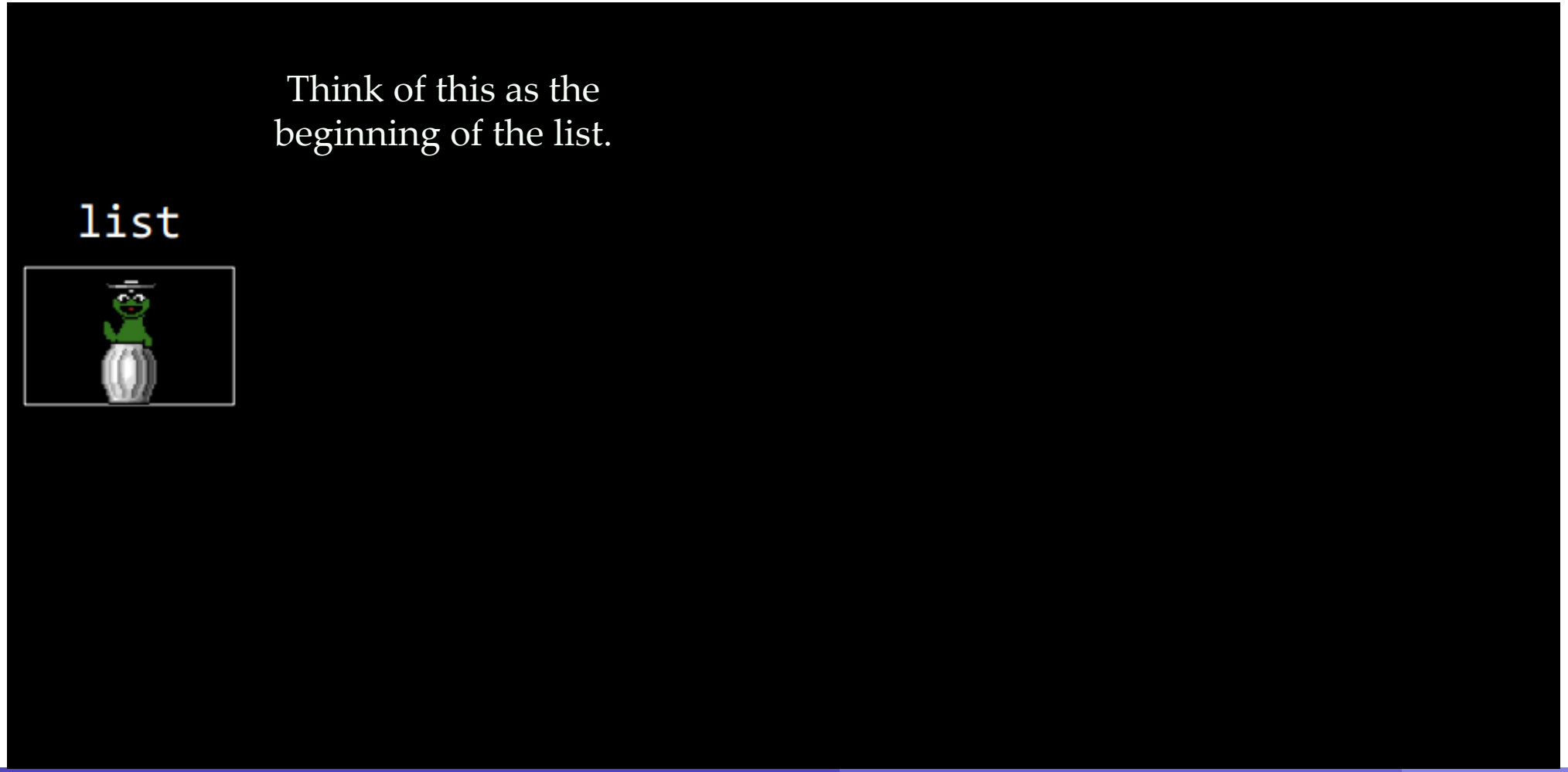
```
typedef struct node
{
    int value;
    struct node* next;
} node;
```



Singly Linked List

Creating a node

```
node* list;
```



Singly Linked List

Creating a node

```
node* list = NULL;
```

Singly Linked List

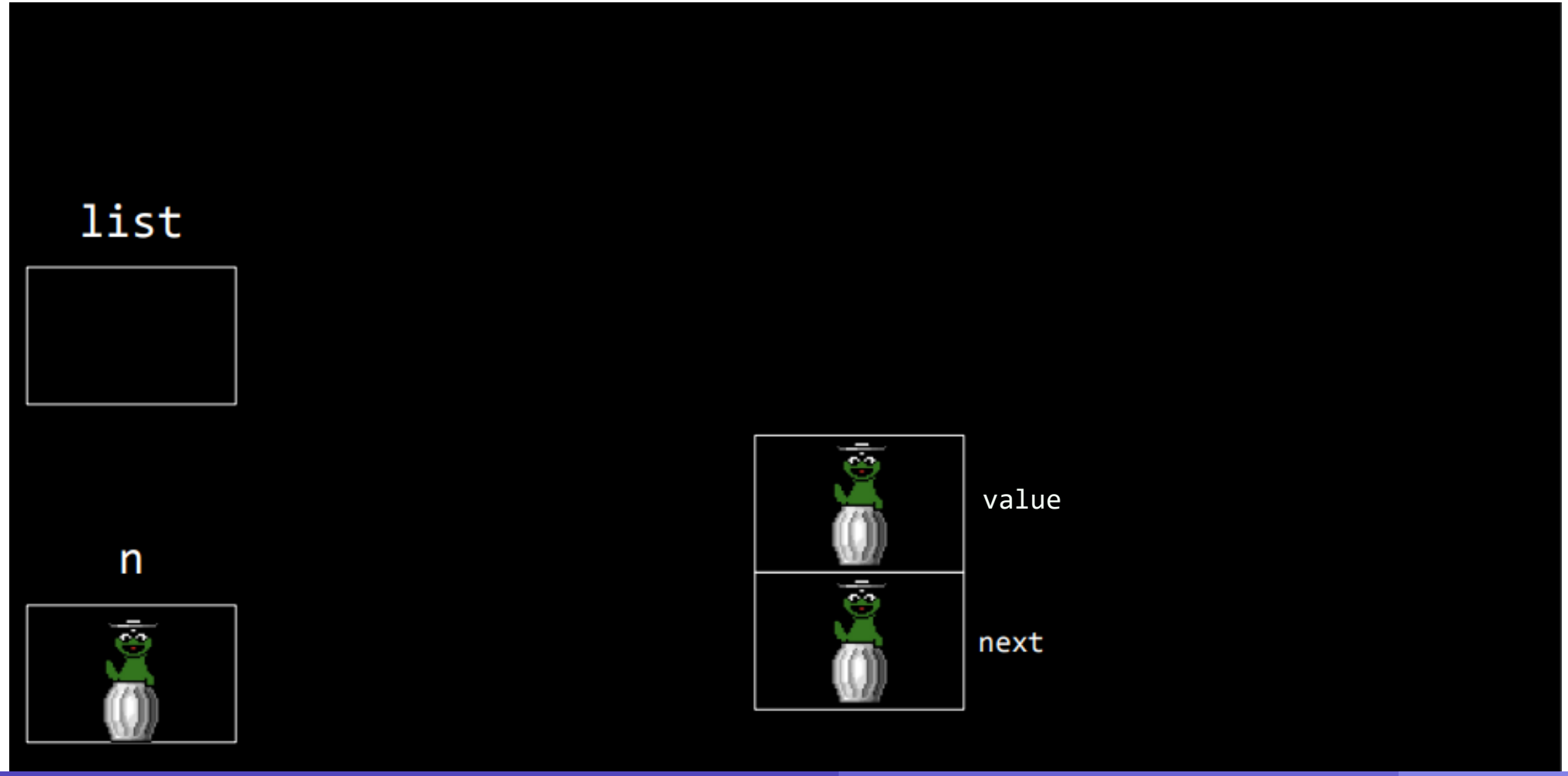
Creating a node

```
node* n = malloc(sizeof(node));
```

Singly Linked List

Creating a node

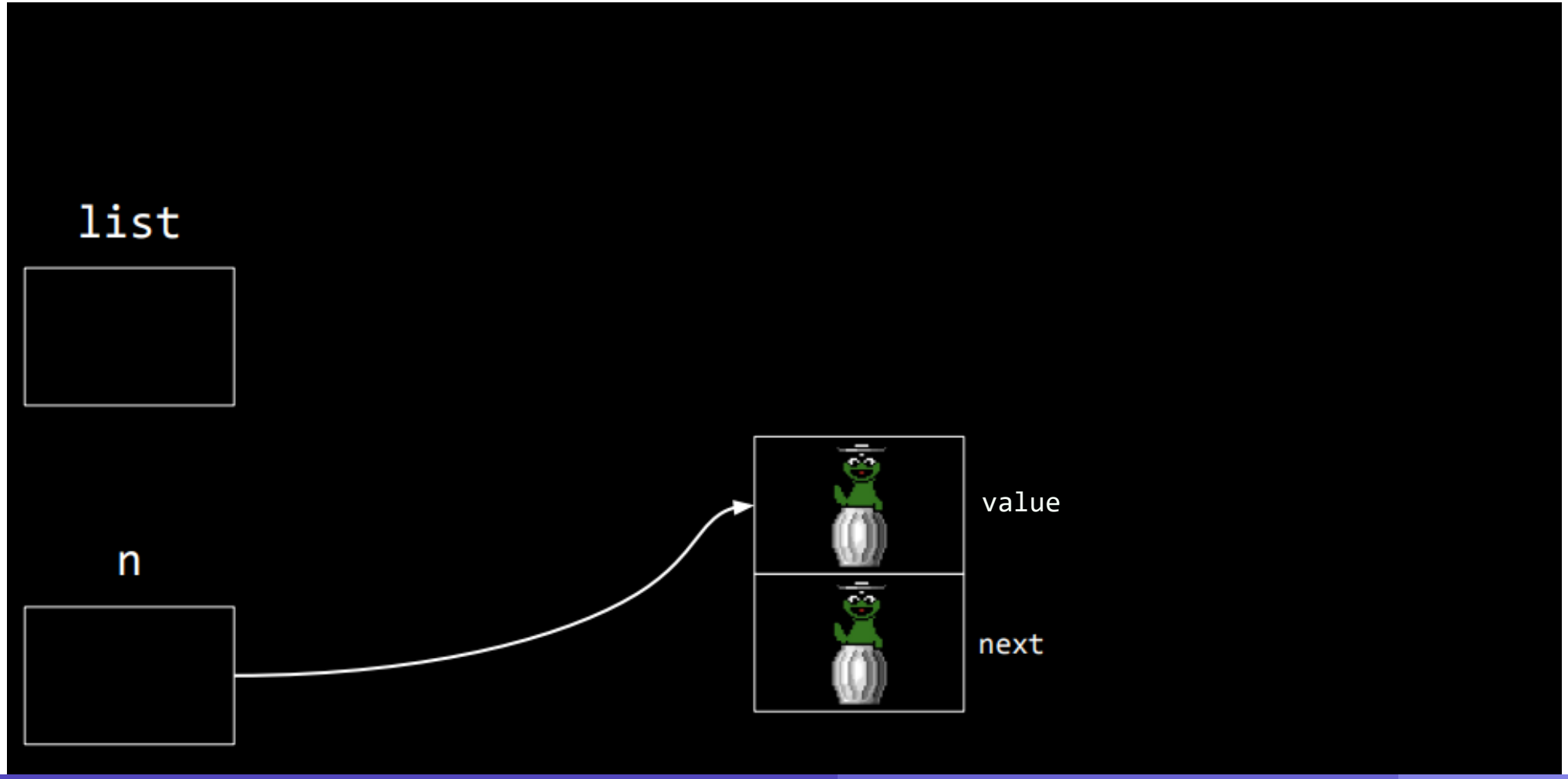
```
node* n = malloc(sizeof(node));
```



Singly Linked List

Creating a node

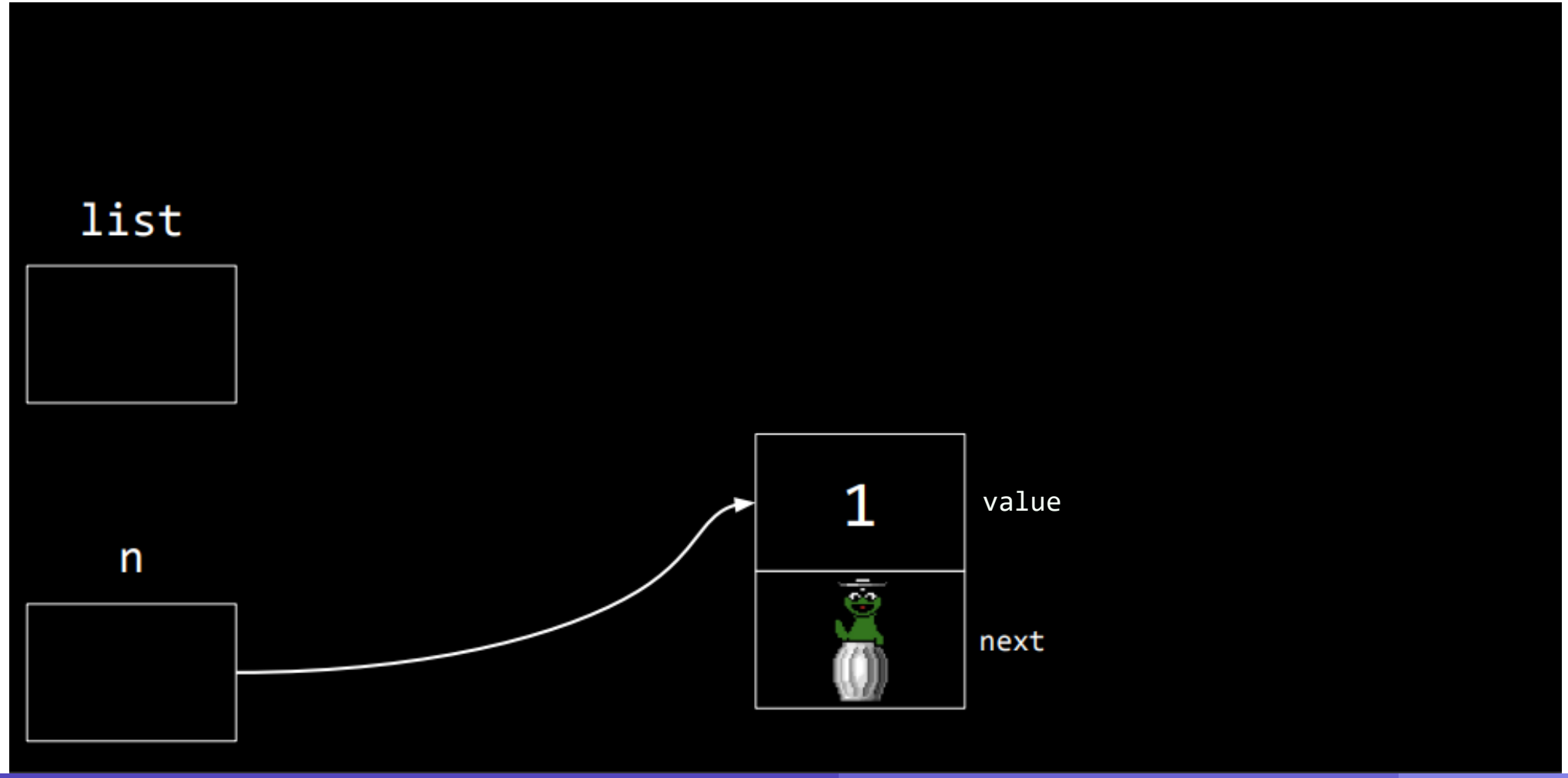
```
node* n = malloc(sizeof(node));
```



Singly Linked List

Creating a node

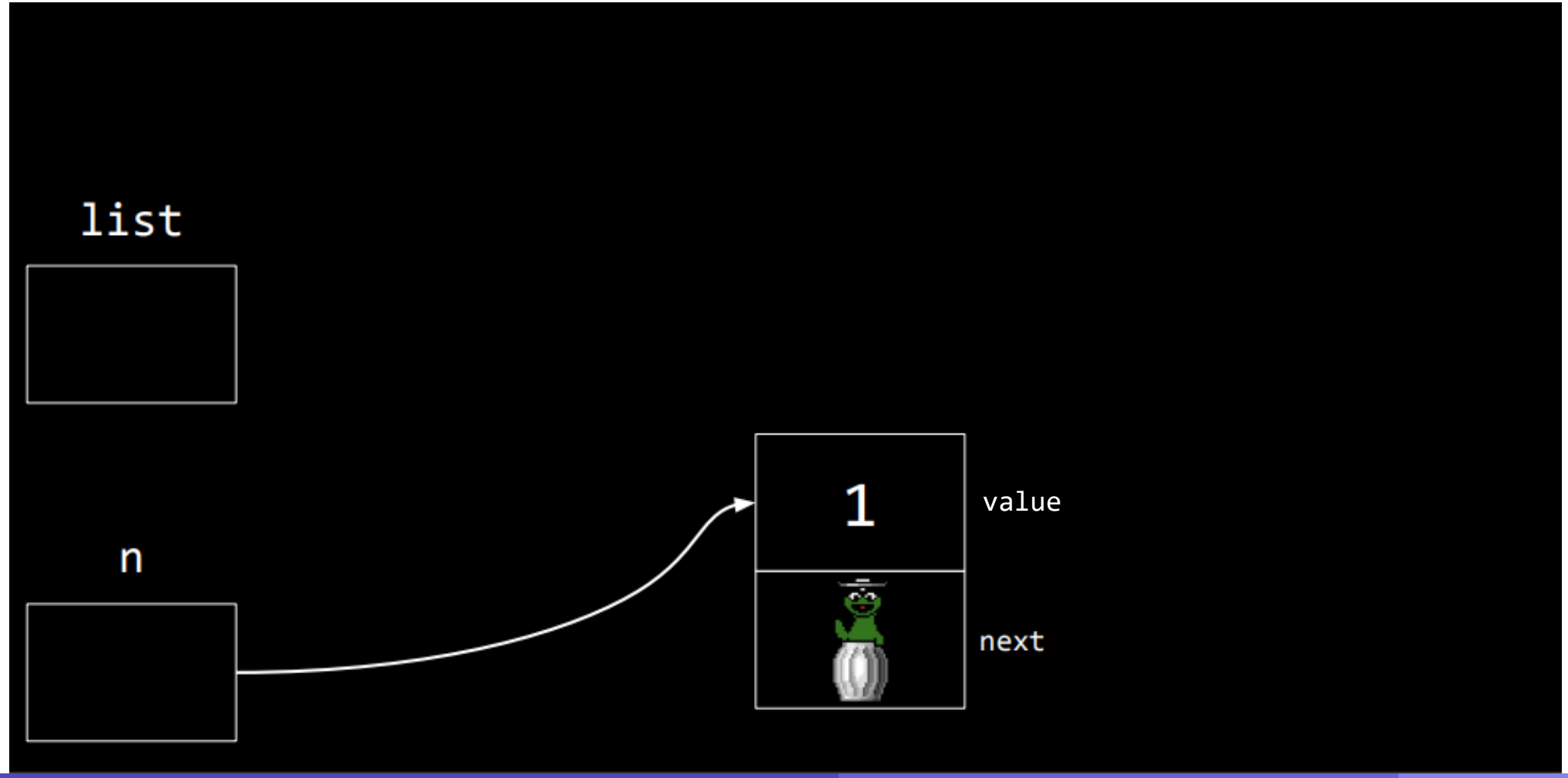
```
(*n).value = 1;
```



Singly Linked List

Creating a node

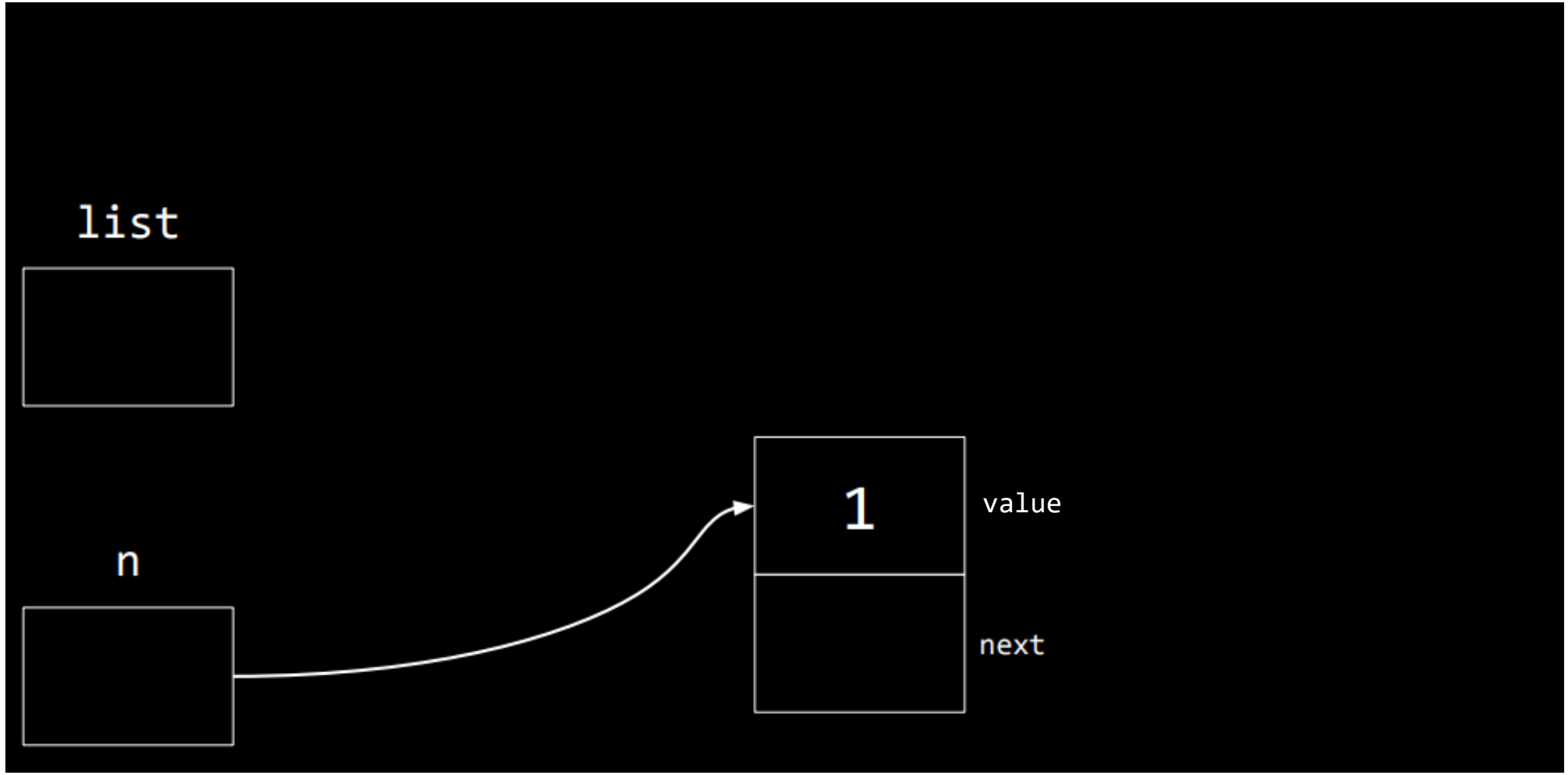
```
n->value = 1;
```



Singly Linked List

Creating a node

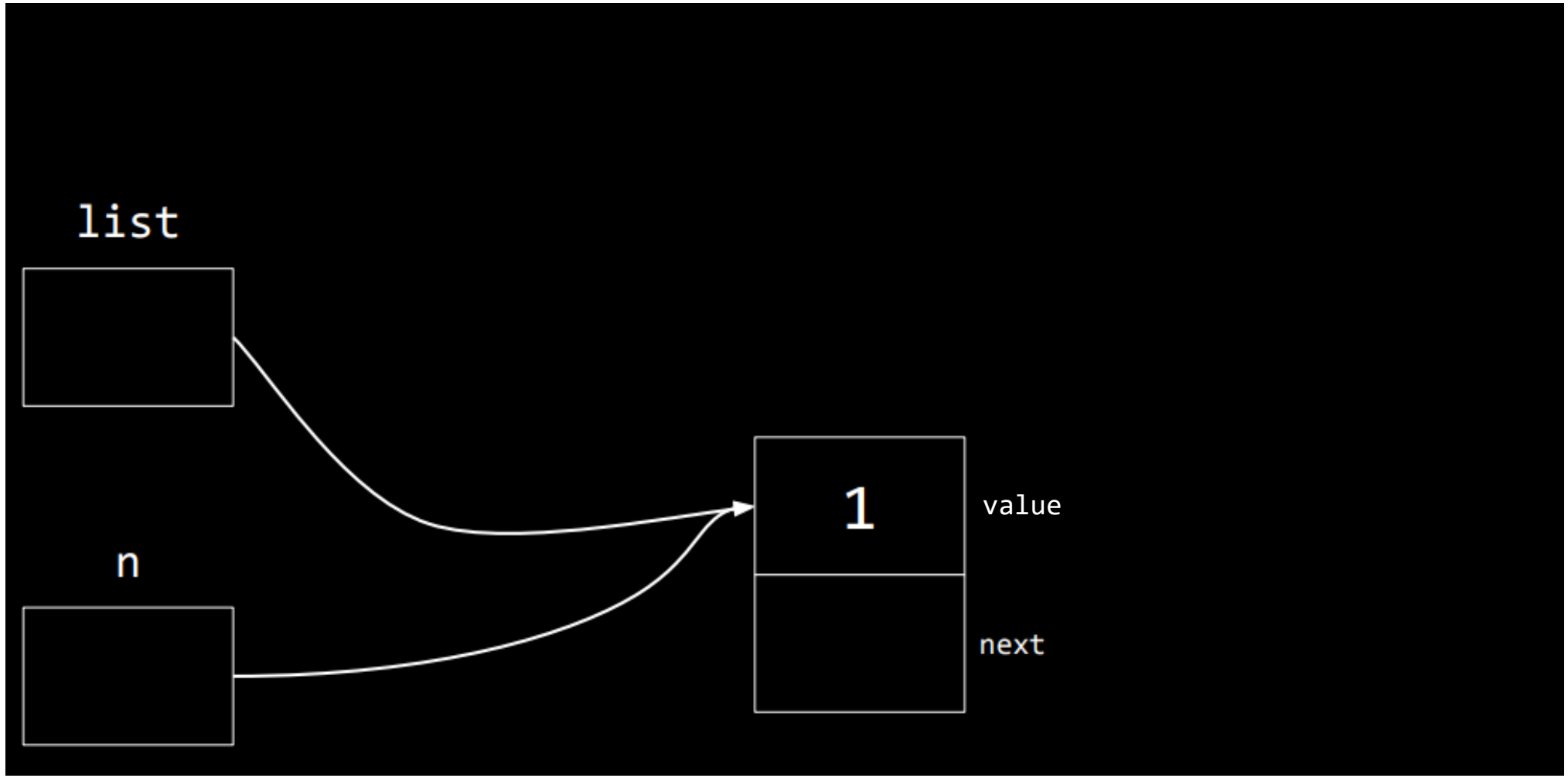
```
n->next = NULL;
```



Singly Linked List

Creating a node

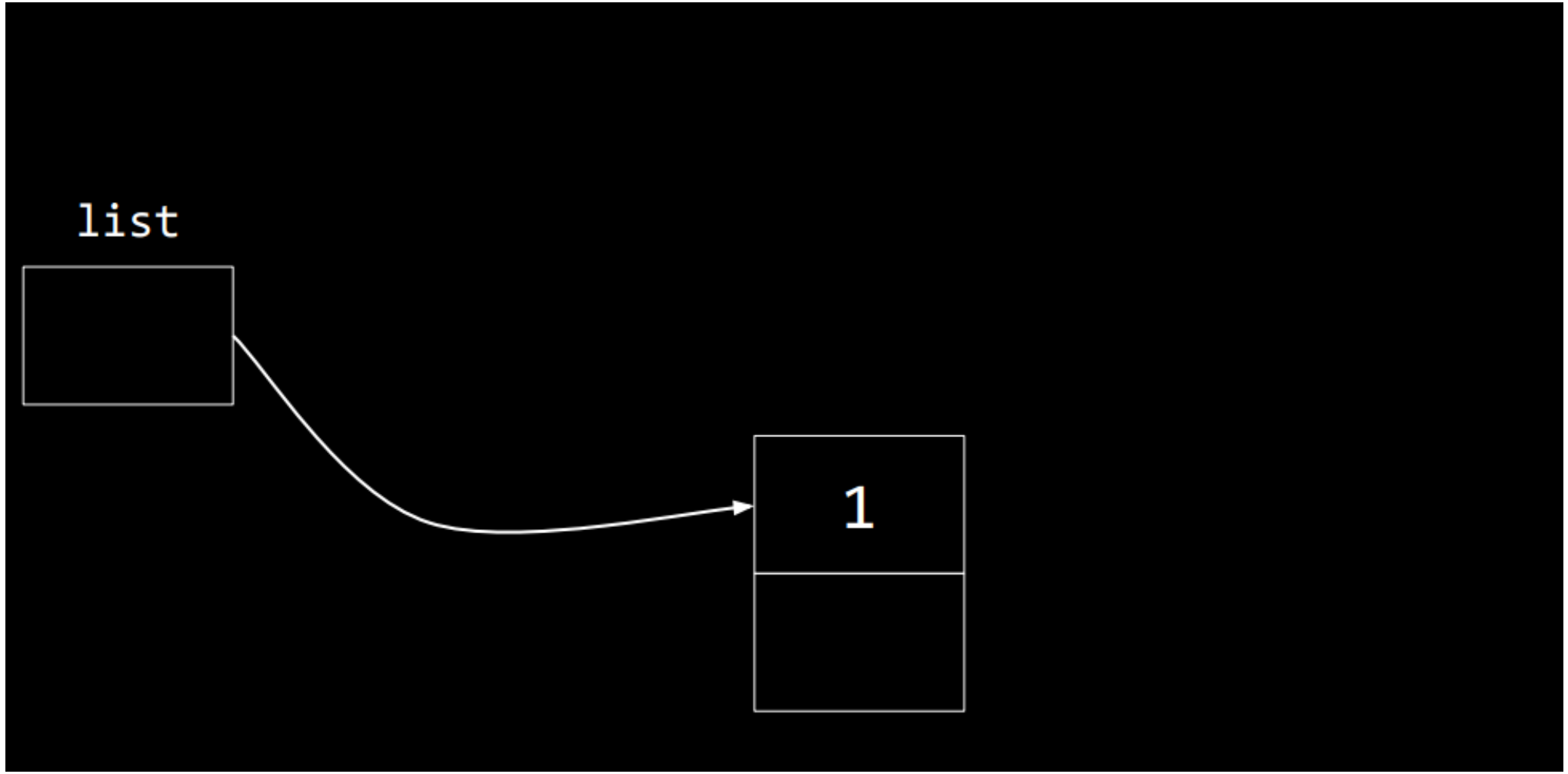
```
list = n;
```



Singly Linked List

Creating a node

Disregarding the node **n**, we end up with a list of size 1.

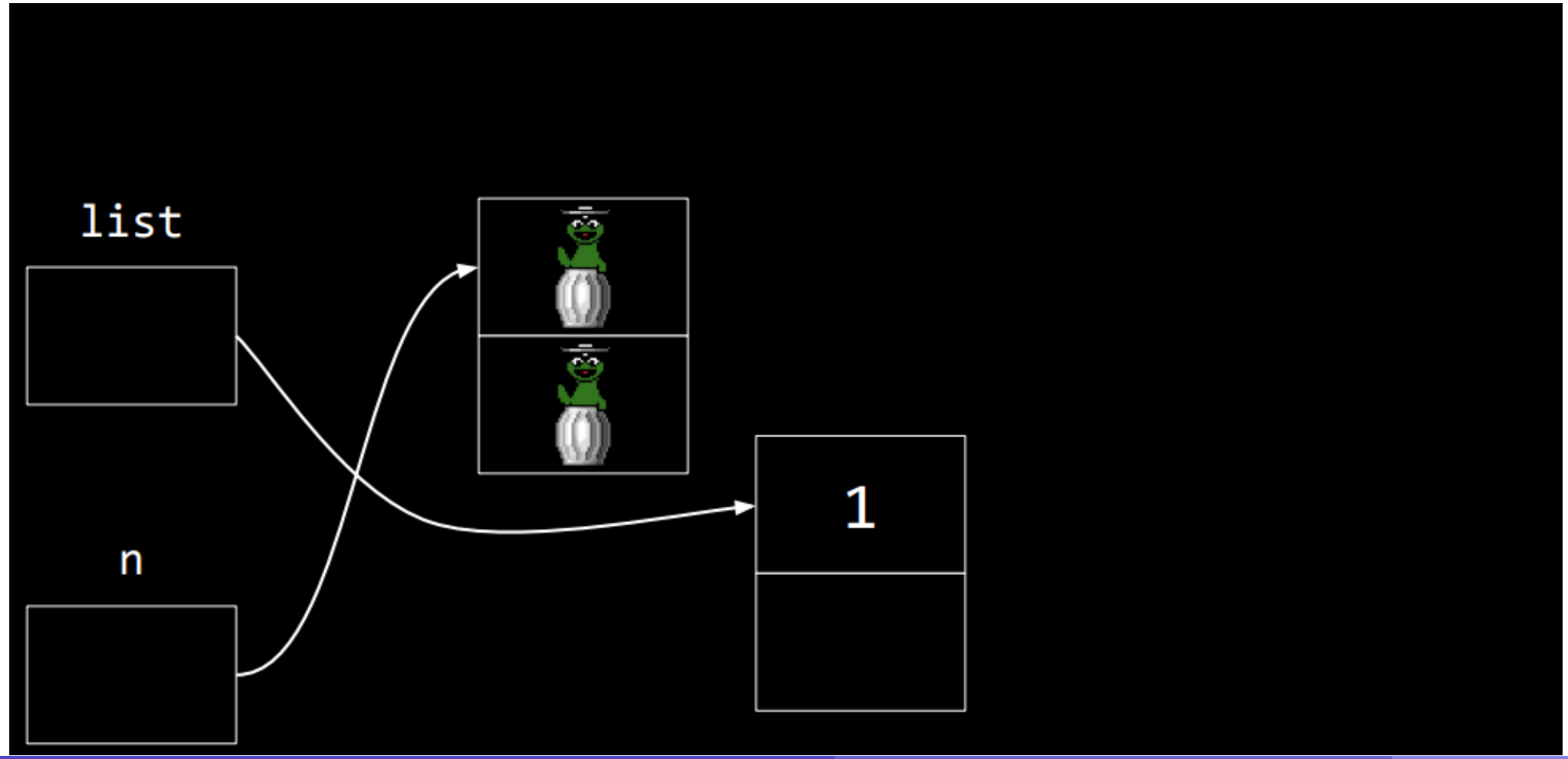


Singly Linked List

Inserting a node (*prepending*)

Let's say we want to add another node to the list.

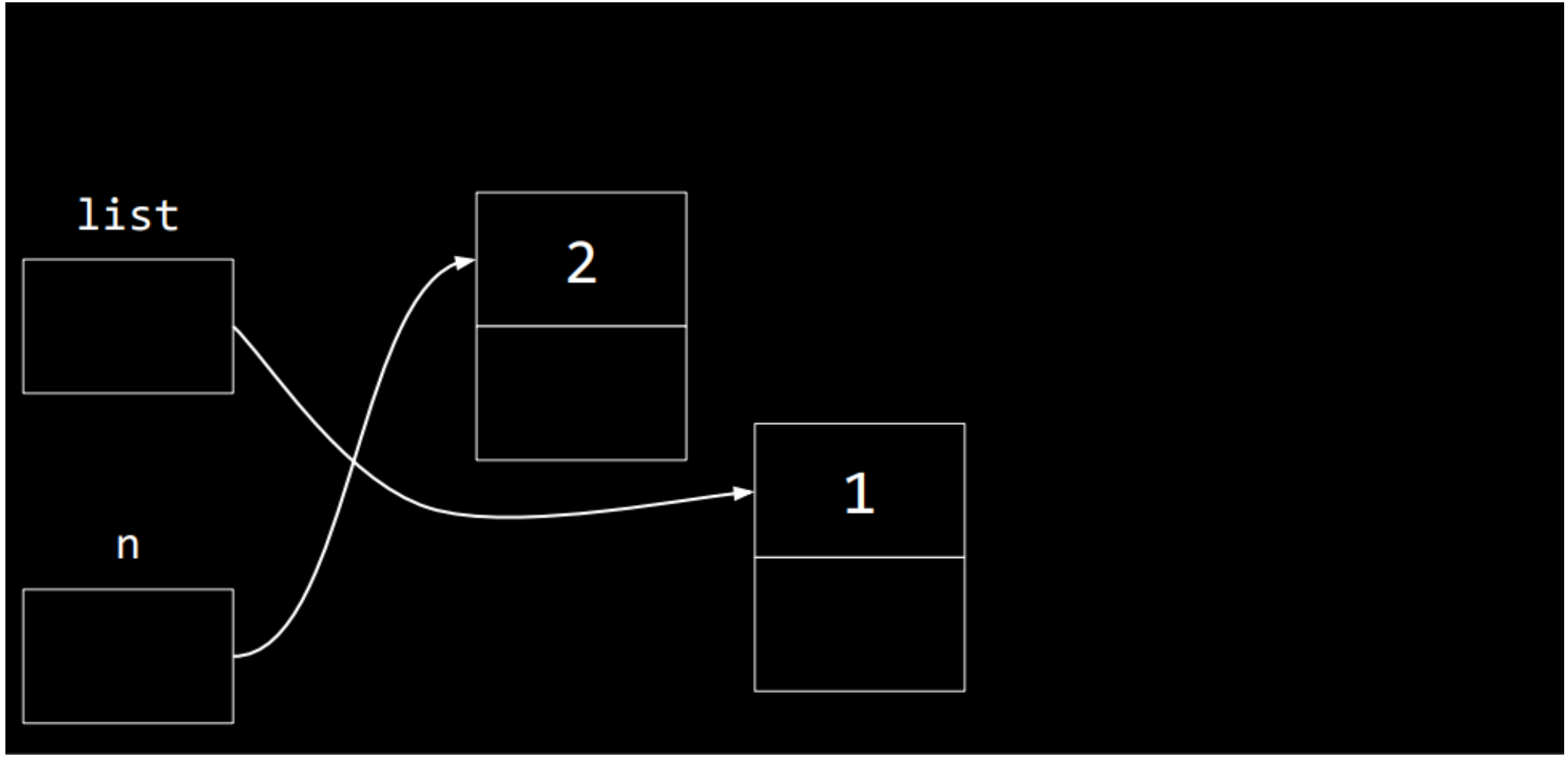
```
node* n = malloc(sizeof(node));
```



Singly Linked List

Inserting a node (*prepending*)

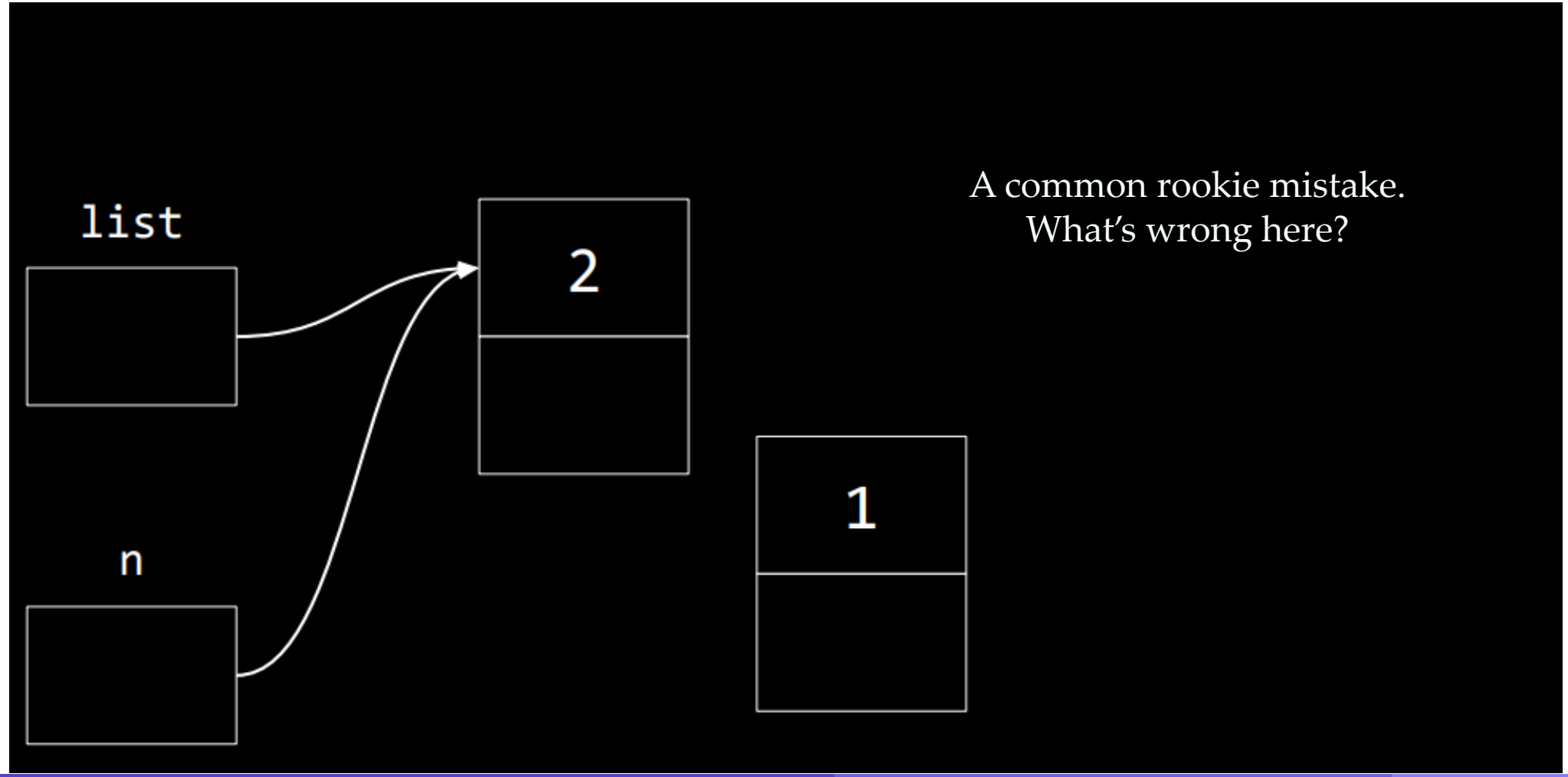
```
n->value = 2;
n->next = NULL;
```



Singly Linked List

Inserting a node (*prepending*)

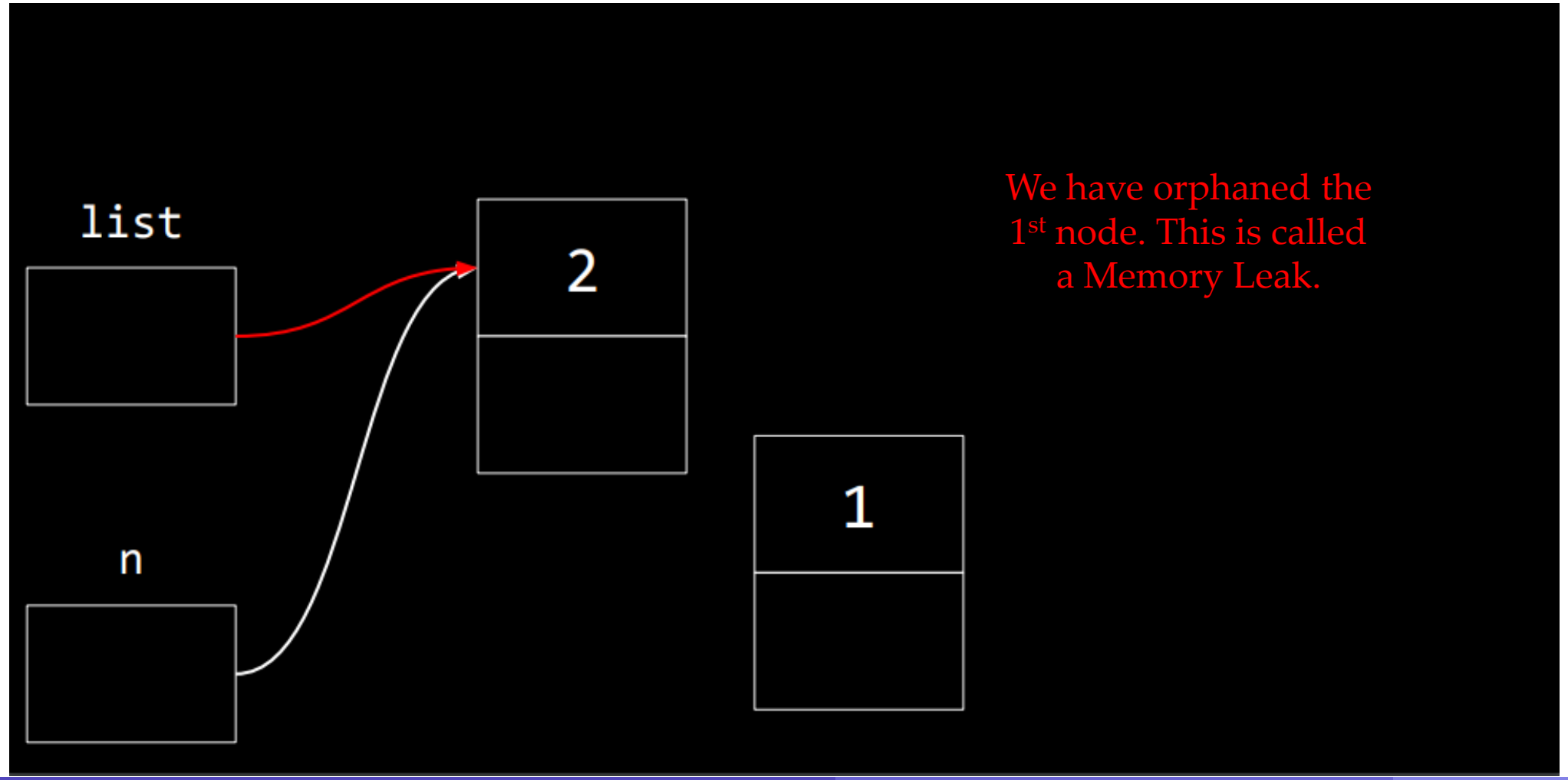
```
list = n;
```



Singly Linked List

Inserting a node (*prepending*)

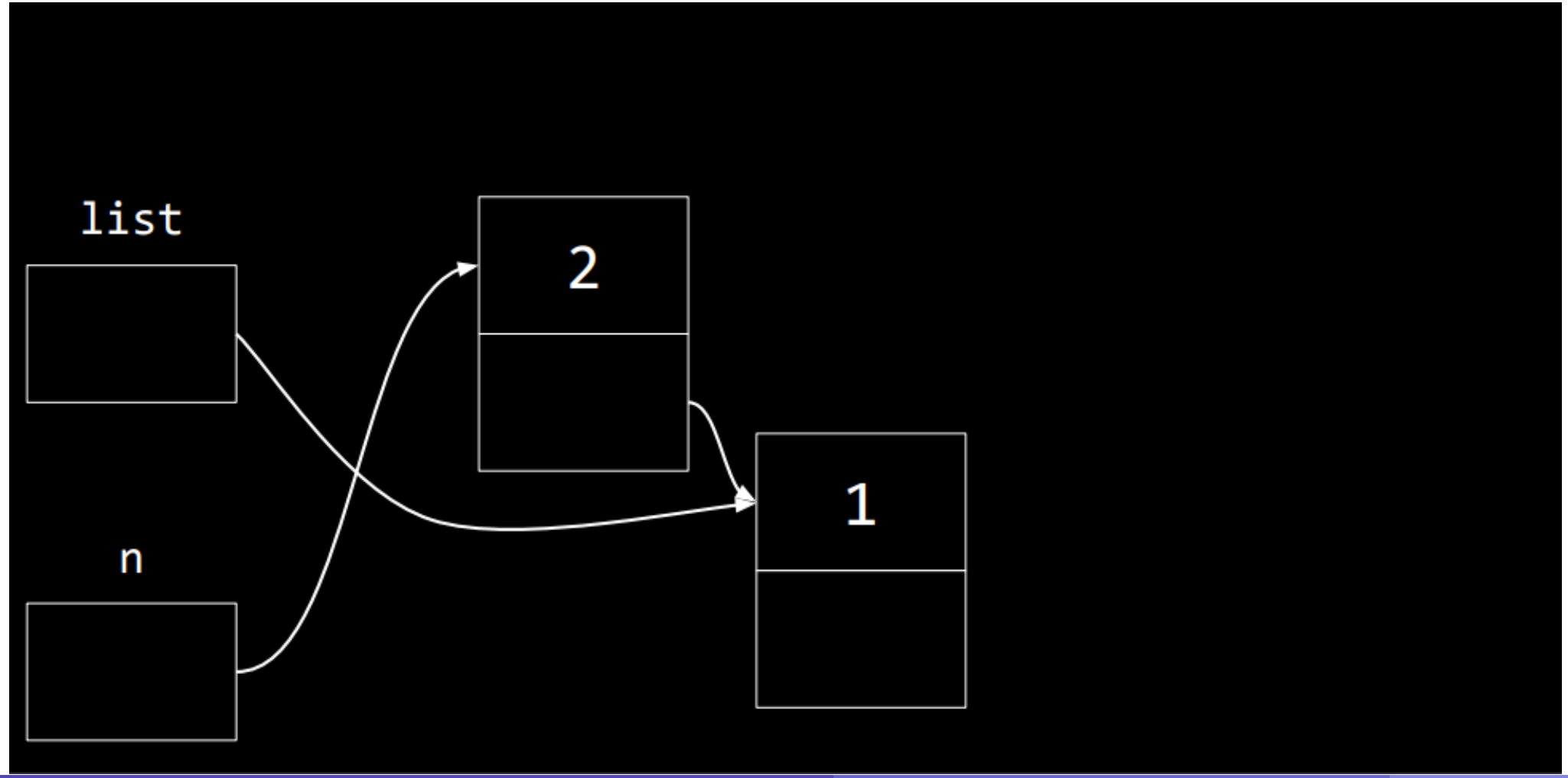
```
list = n;
```



Singly Linked List

Inserting a node (*prepending*)

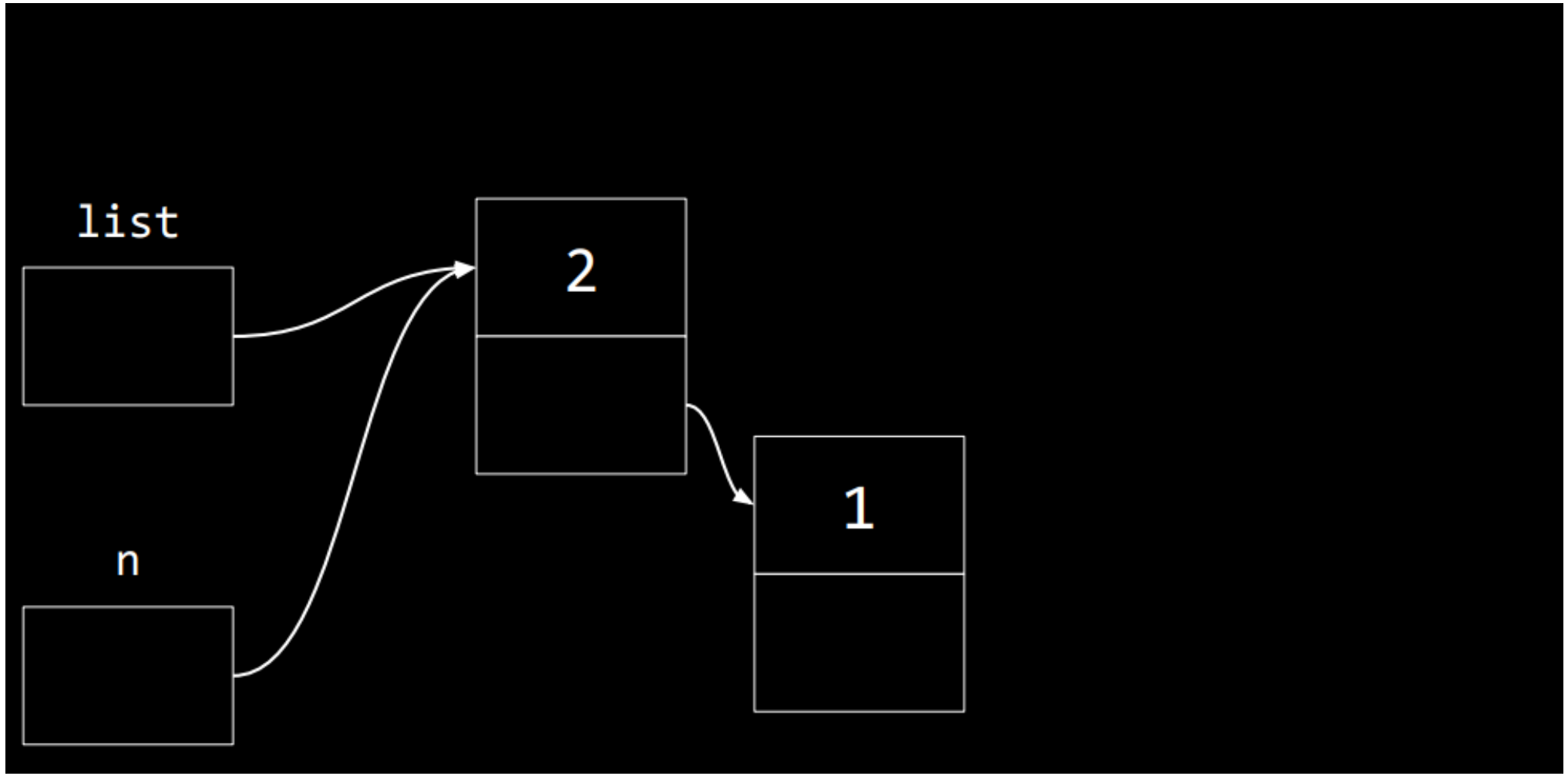
```
n->next = list;
```



Singly Linked List

Inserting a node (*prepending*)

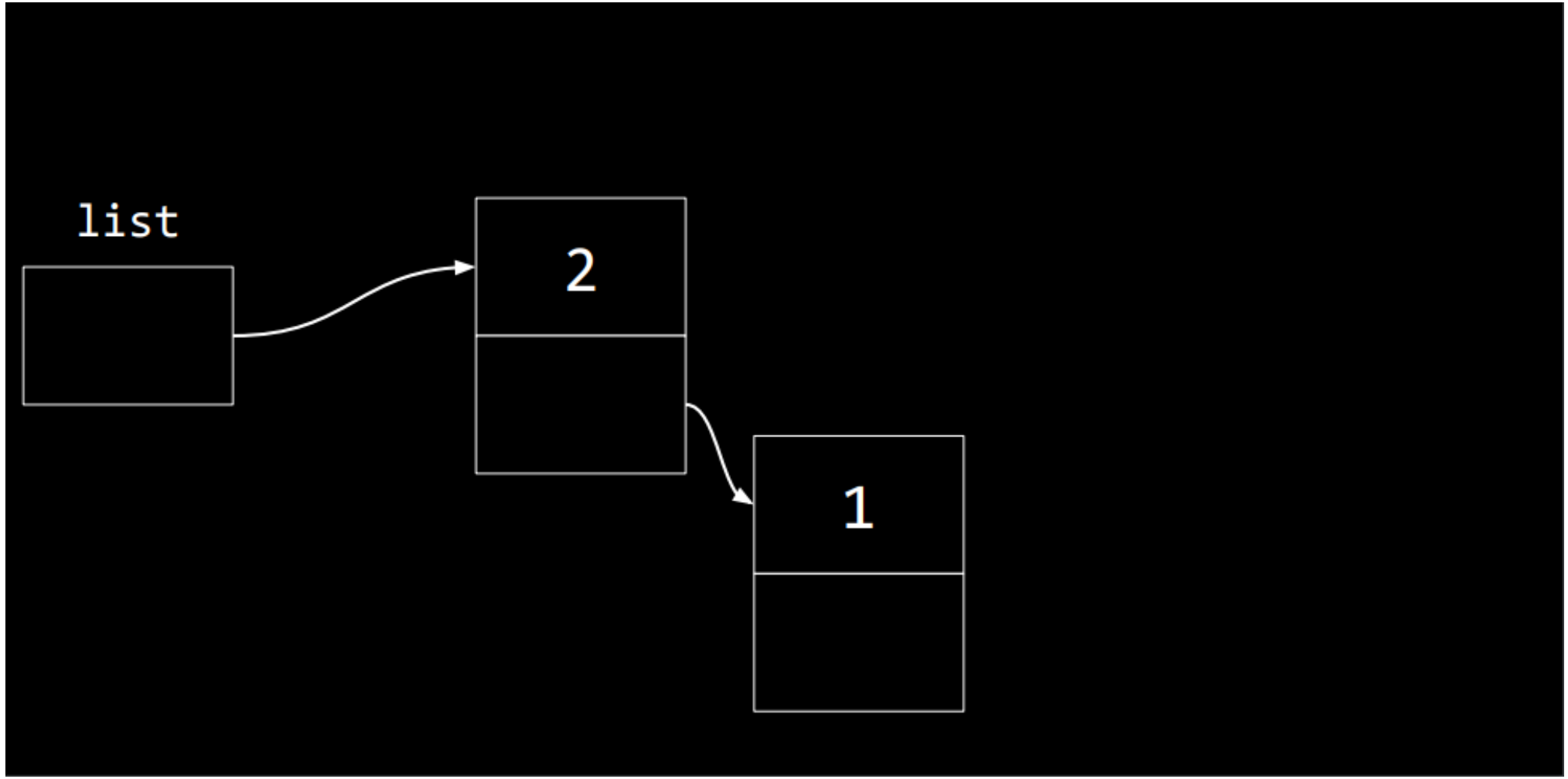
```
list = n;
```



Singly Linked List

Inserting a node (*prepending*)

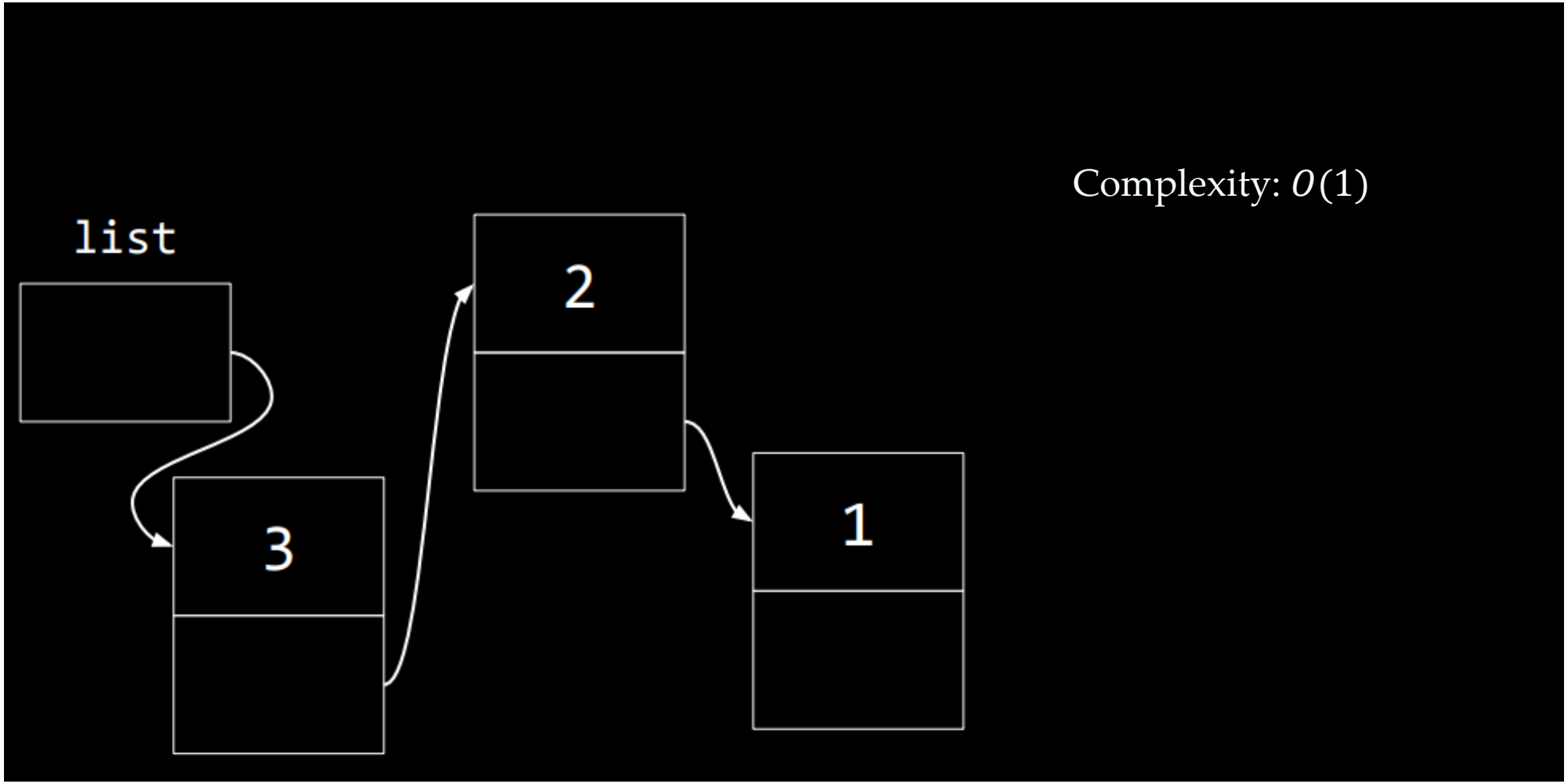
Disregarding the node **n**, we end up with a list of size 2.



Singly Linked List

Inserting a node (*prepending*)

After adding another node having the value 3.

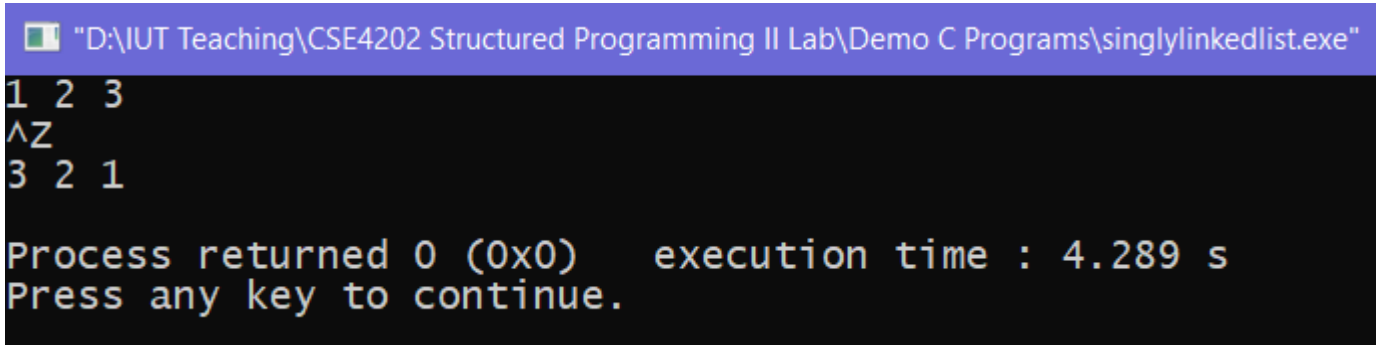


Singly Linked List

Inserting a node (*prepending*)

The C implementation is,

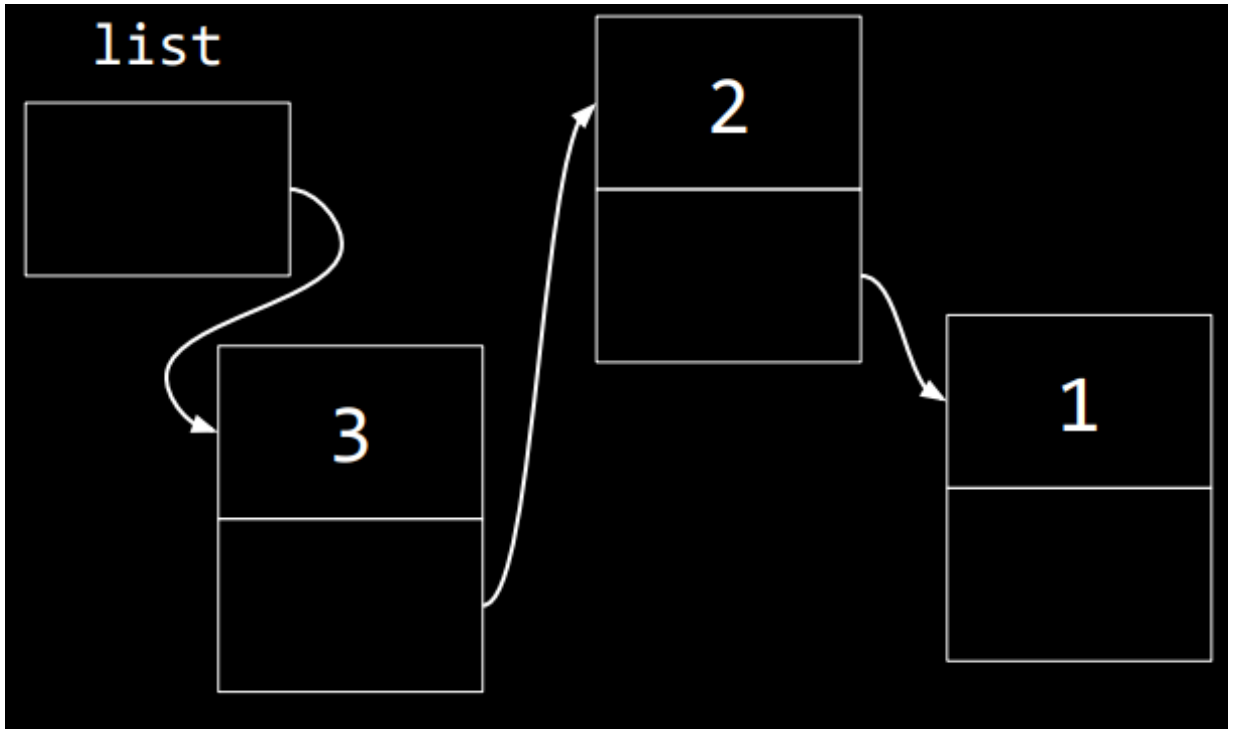
```
void prepend(int element)
{
    node* n = (node*)malloc(sizeof(node));
    n->value = element;
    n->next = NULL;
    n->next = list;
    list = n;
}

int main()
{
    int x;
    while(scanf("%d", &x) != EOF)
    {
        prepend(x);
    }
    print_list();
    ...
    ...
    ...
```



Singly Linked List

Printing the contents of the list

We traverse through the list and print the values.



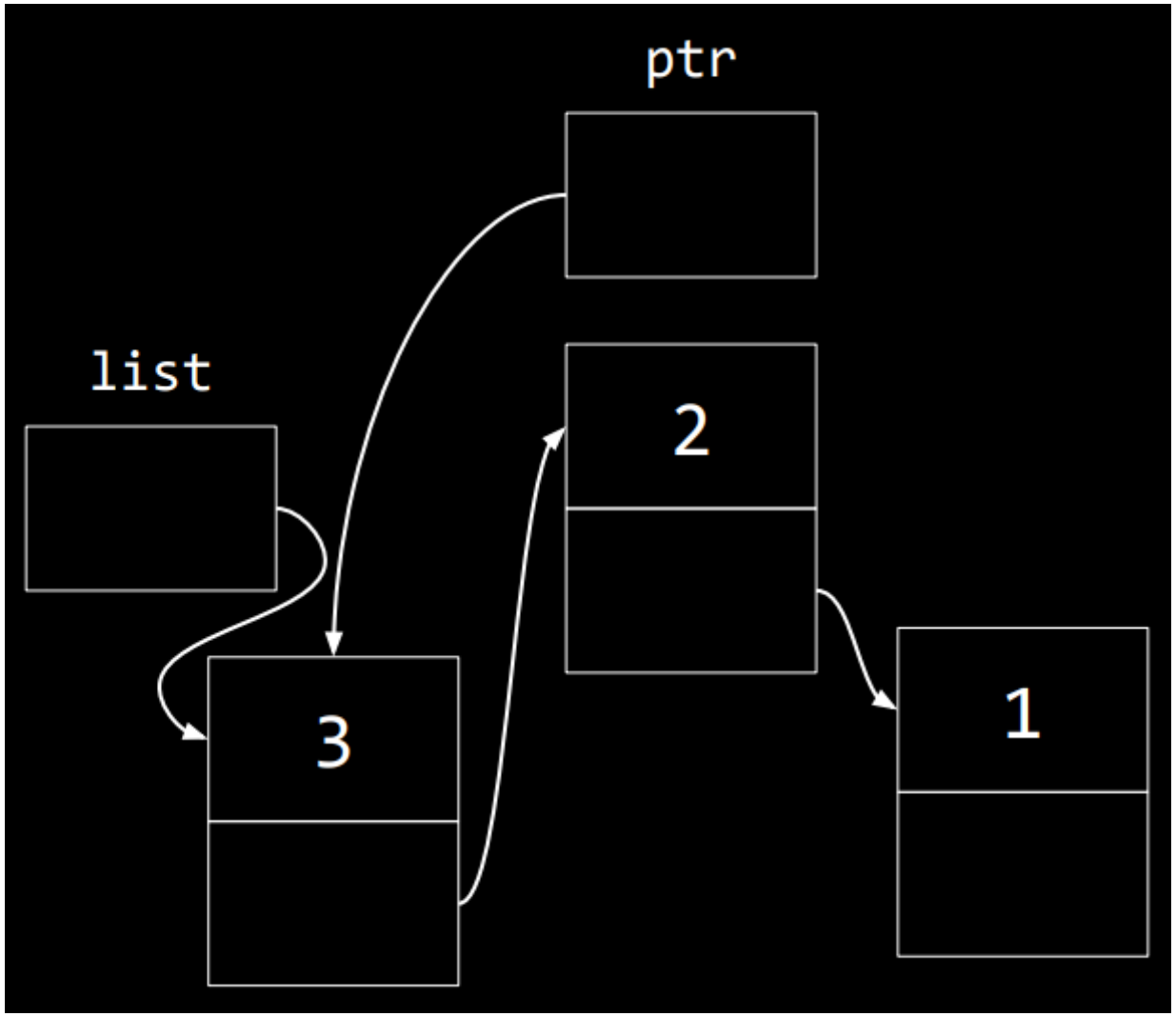

```
void print_list()
{
    if(list == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    for(node* ptr = list; ptr != NULL; ptr = ptr->next)
    {
        printf("%d ", ptr->value);
    }
    printf("\n");
}
```

Singly Linked List

Printing the contents of the list

We traverse through the list and print the values.

```
void print_list()
{
    if(list == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    for(node* ptr = list; ptr != NULL; ptr = ptr->next)
    {
        printf("%d ", ptr->value);
    }
    printf("\n");
}
```

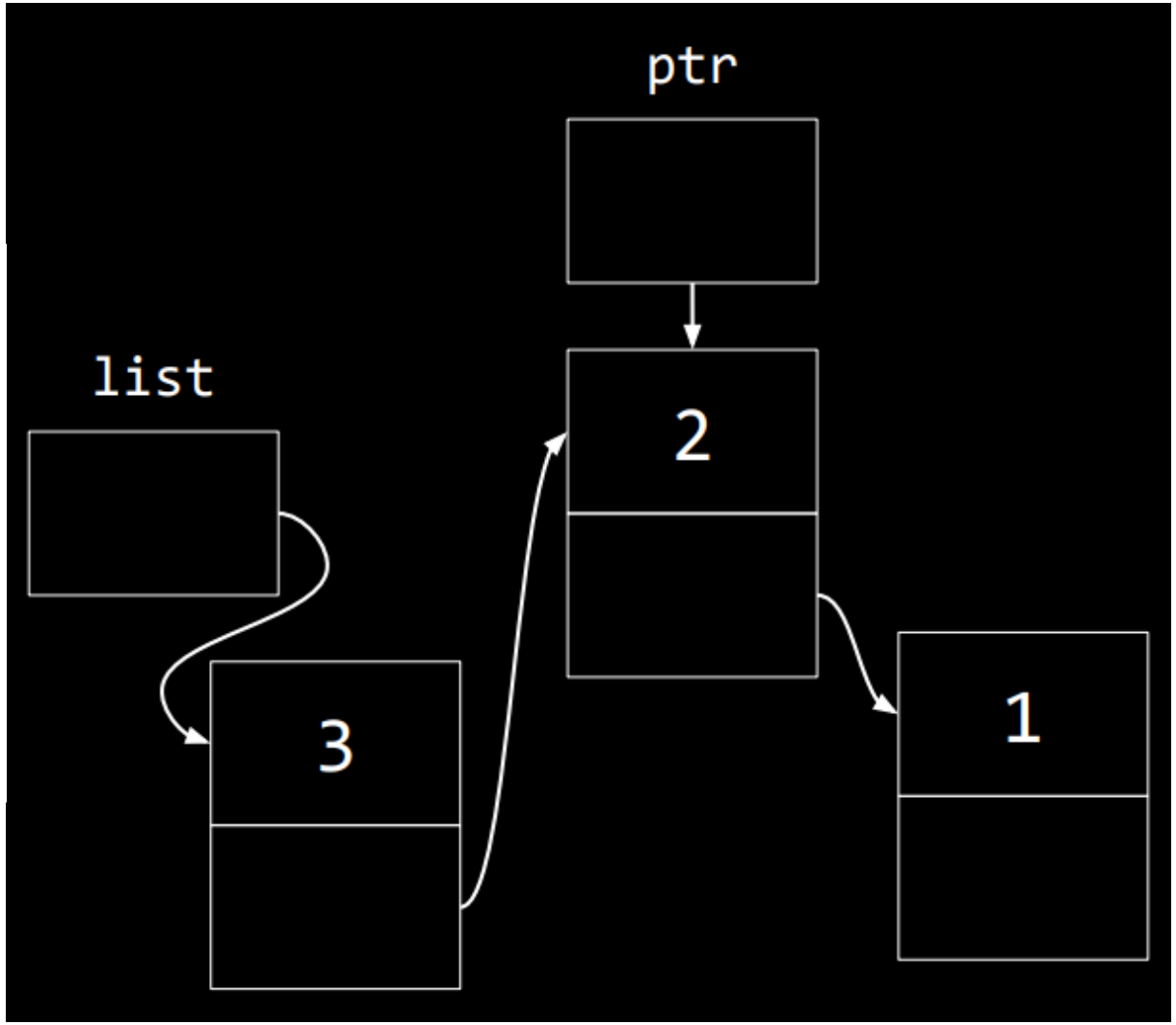


Singly Linked List

Printing the contents of the list

We traverse through the list and print the values.

```
void print_list()
{
    if(list == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    for(node* ptr = list; ptr != NULL; ptr = ptr->next)
    {
        printf("%d ", ptr->value);
    }
    printf("\n");
}
```



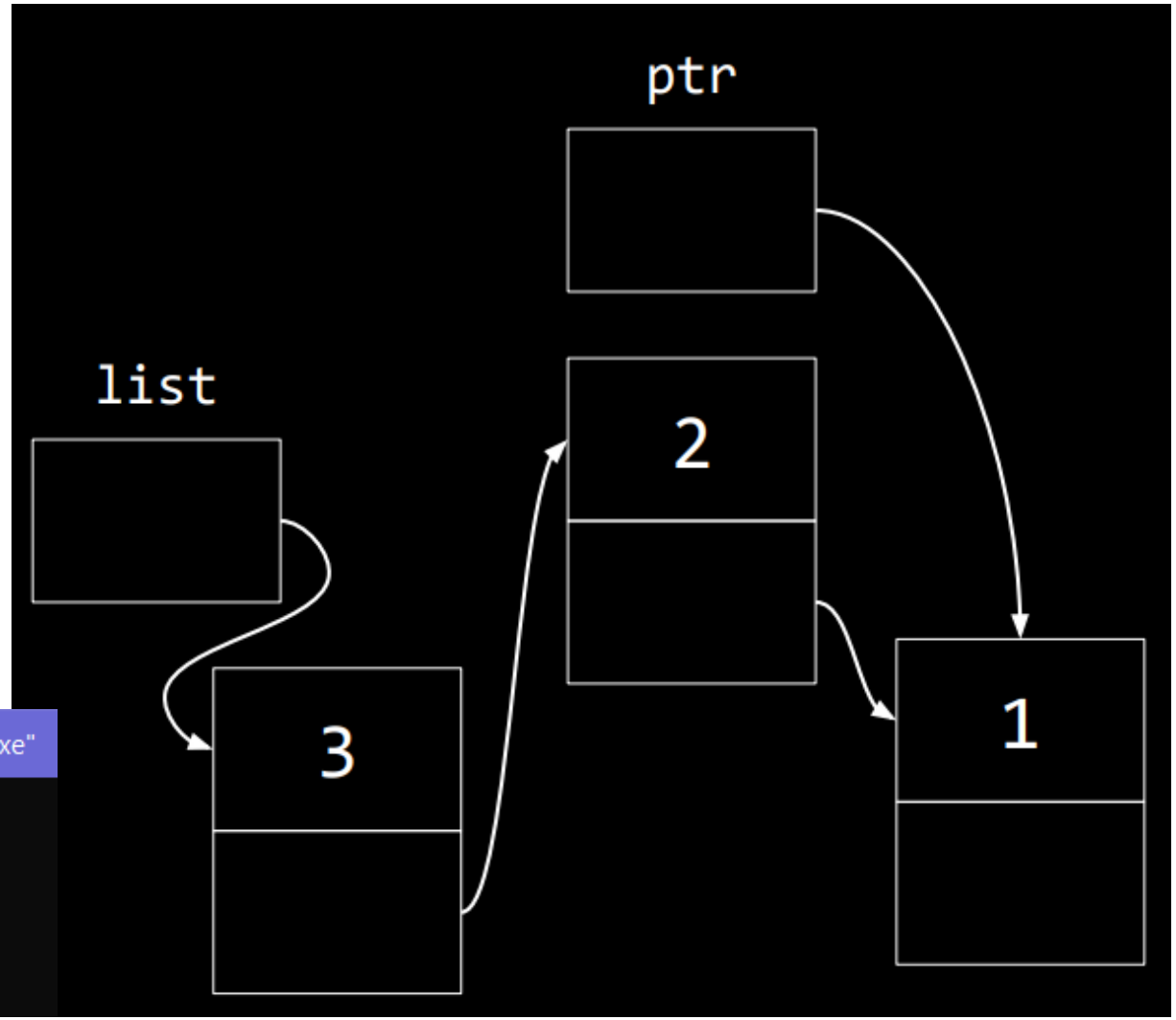
Singly Linked List

Printing the contents of the list

What are we sacrificing here?
There are no indices!

We traverse through the list and print the values.

```
void print_list()
{
    if(list == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    for(node* ptr = list; ptr != NULL; ptr = ptr->next)
    {
        printf("%d ", ptr->value);
    }
    printf("\n");
}
```

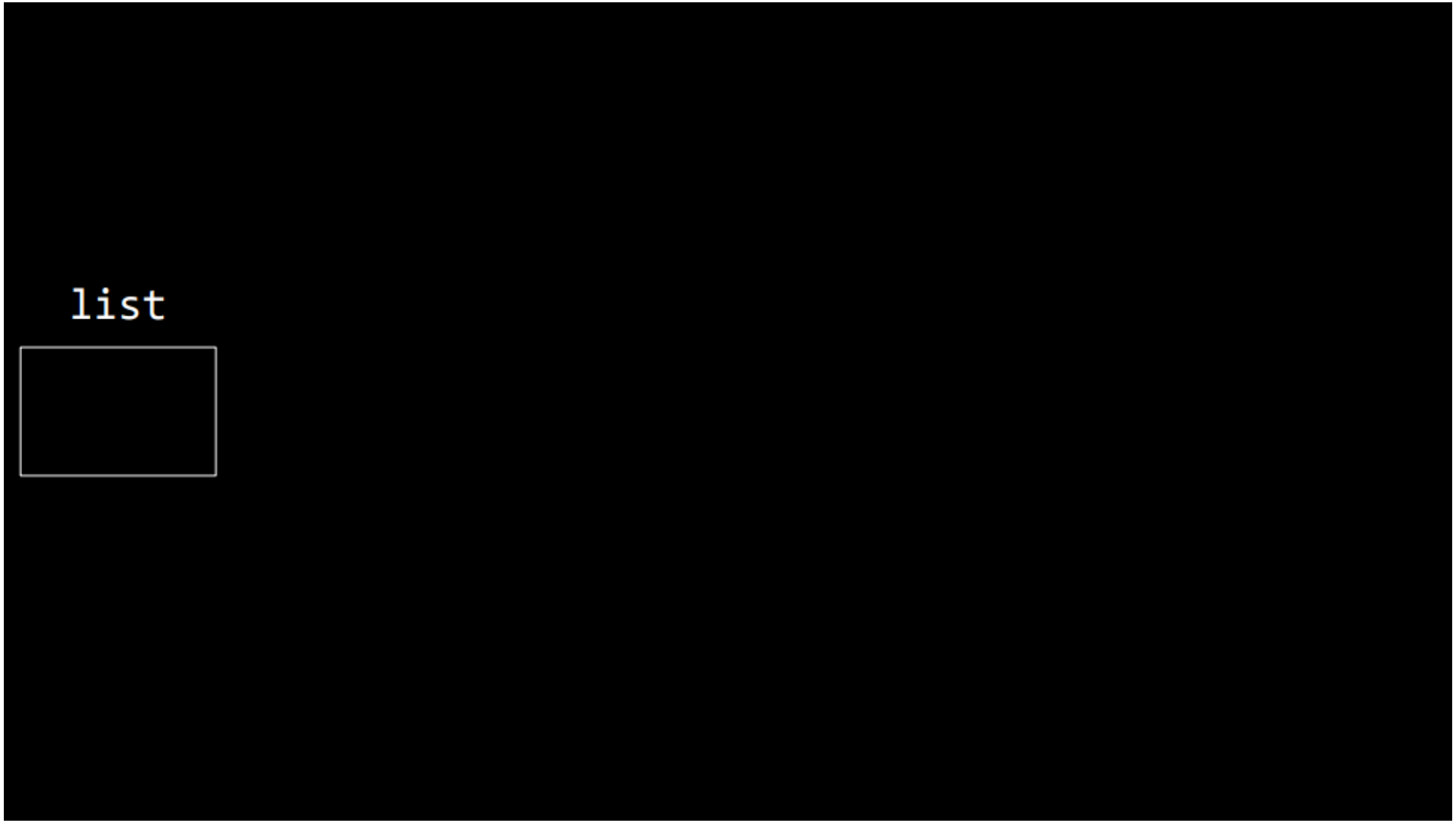



```
"D:\IUT Teaching\CSE4202 Structured Programming II Lab\Demo C Programs\singlylinkedlist.exe"
1 2 3
^Z
3 2 1

Process returned 0 (0x0)   execution time : 4.479 s
Press any key to continue.
```

Singly Linked List

Inserting a node (*appending*)

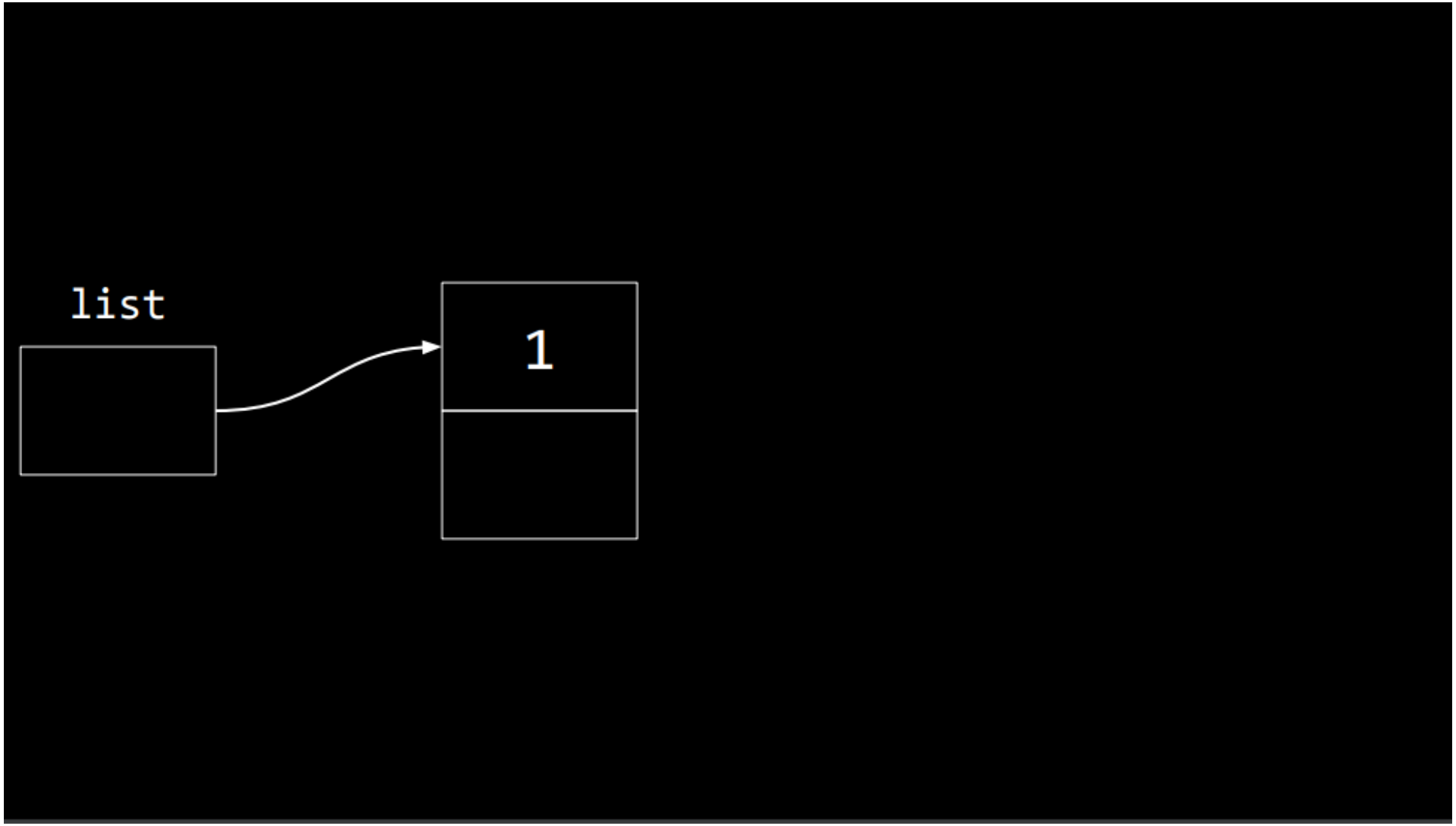
We traverse through the list and add a node at the end.



Singly Linked List

Inserting a node (*appending*)

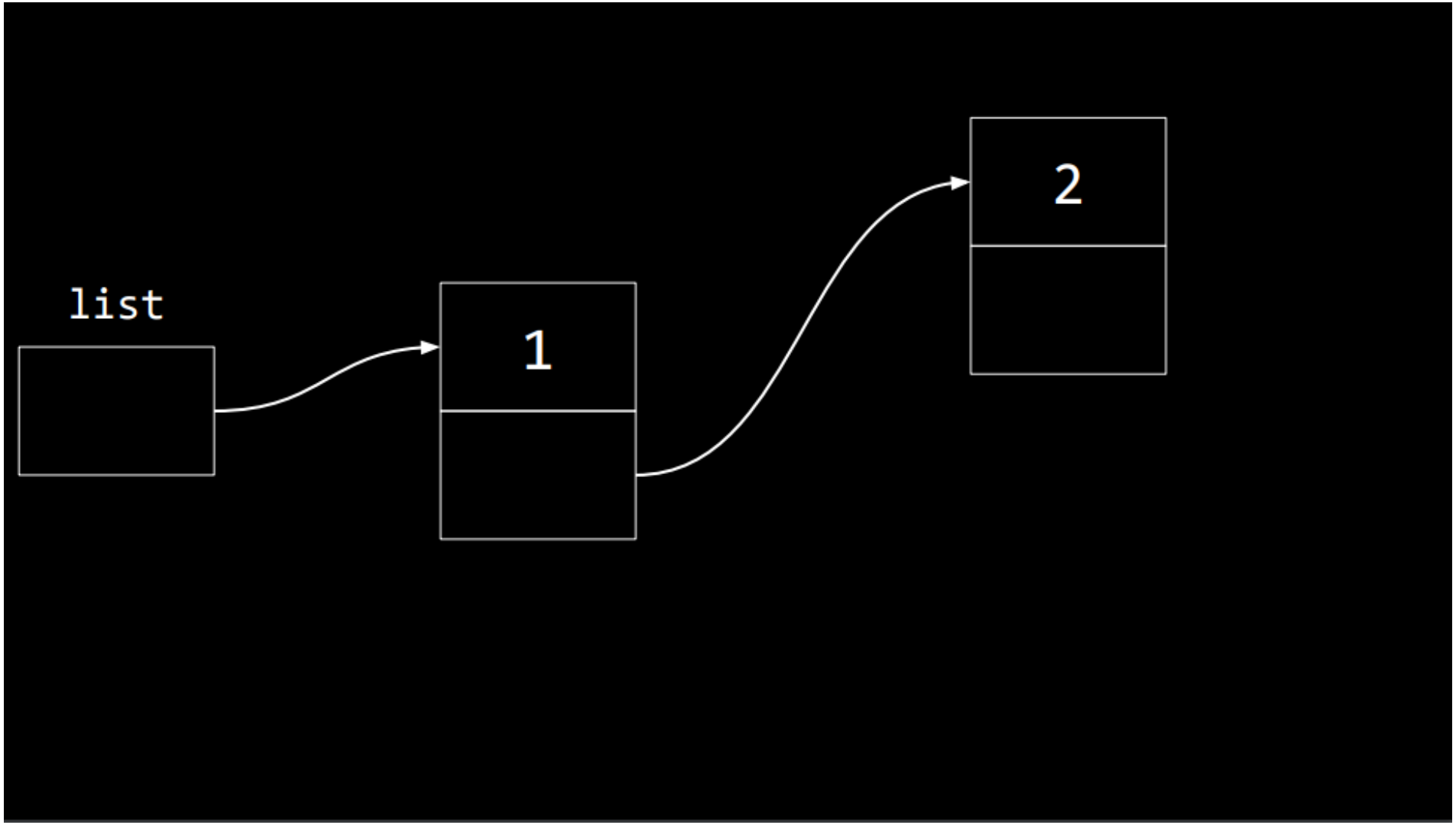
We traverse through the list and add a node at the end.



Singly Linked List

Inserting a node (*appending*)

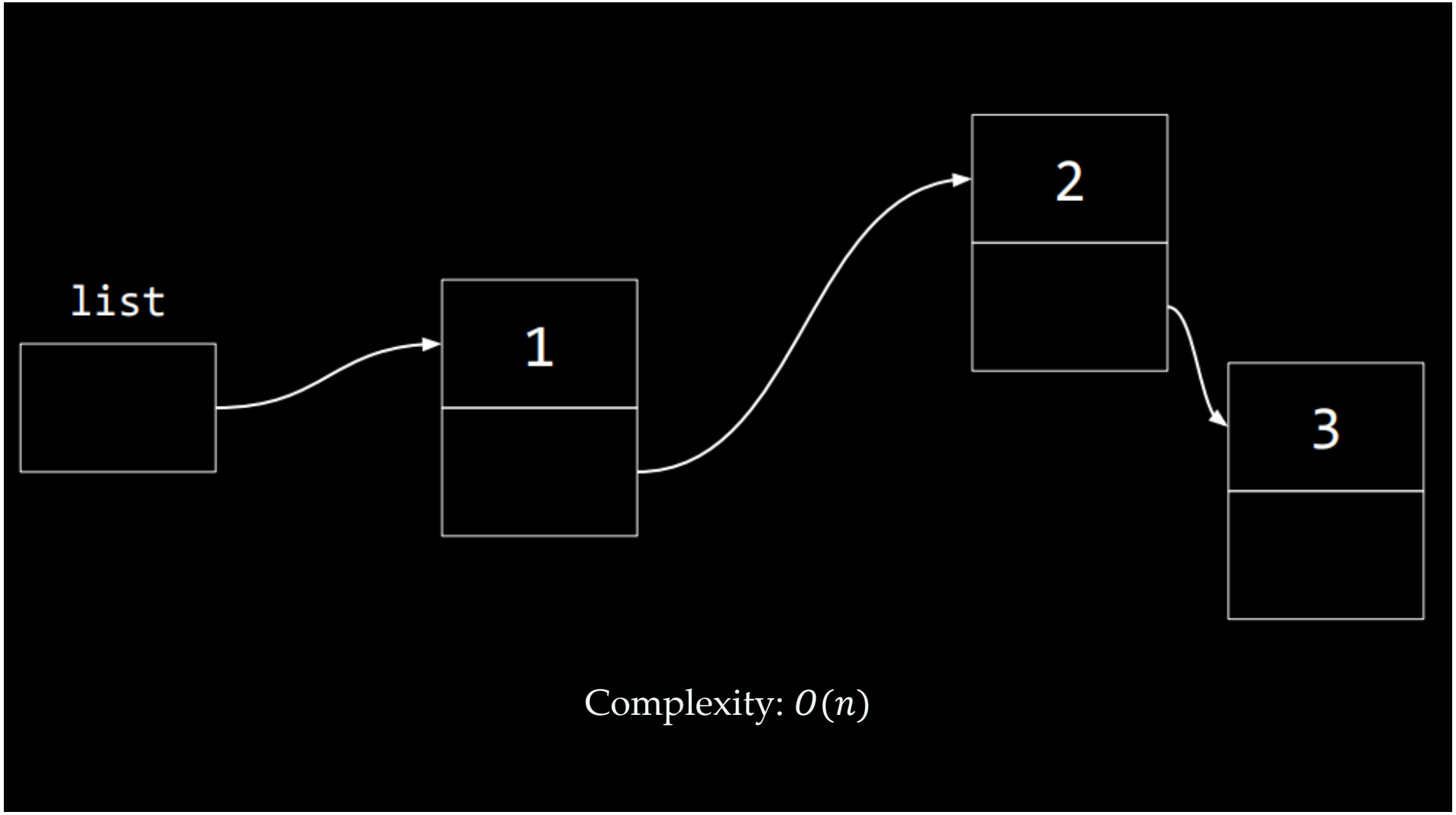
We traverse through the list and add a node at the end.



Singly Linked List

Inserting a node (*appending*)

We traverse through the list and add a node at the end.



Singly Linked List

Inserting a node (*appending*)

We traverse through the list and add a node at the end.

```
void append(int element)
{
    node* n = (node*)malloc(sizeof(node));
    n->value = element;
    n->next = NULL;

    if(list == NULL)
    {
        list = n;
        return;
    }
    node* temp = list;
    while(temp->next != NULL)
    {
        temp = temp->next;
    }
    temp->next = n;
}
```

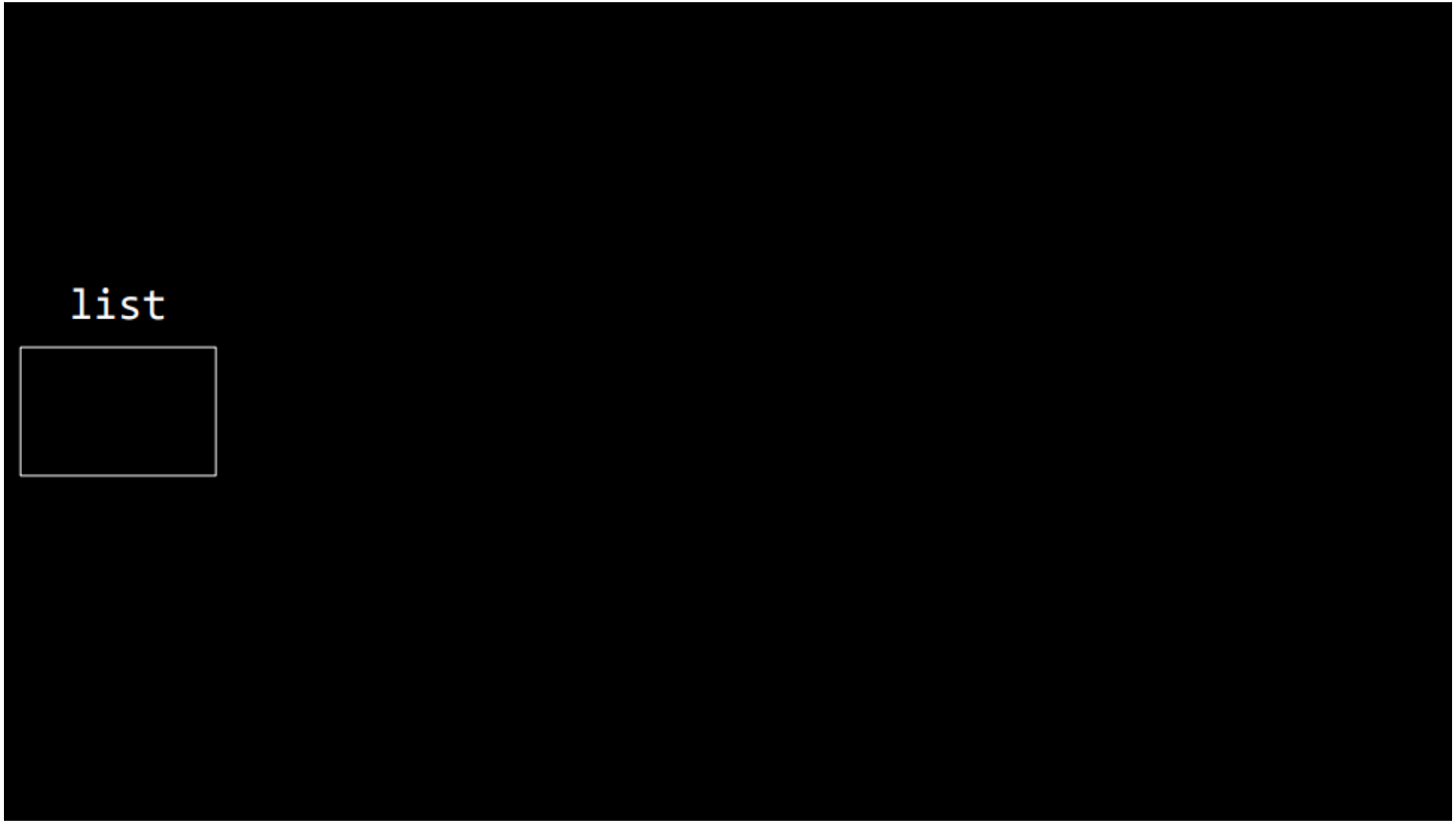
```
int main()
{
    int x;
    while(scanf("%d", &x) != EOF)
    {
        append(x);
    }
    print_list();
    ...
    ...
    ...
```

```
Select "D:\IUT Teaching\CSE4202 Structured Programming II Lab\Demo C Programs\singlylinkedlist.exe"
1 2 3
^Z
1 2 3

Process returned 0 (0x0)   execution time : 5.047 s
Press any key to continue.
```

Singly Linked List

Inserting a node (*sorted order*)

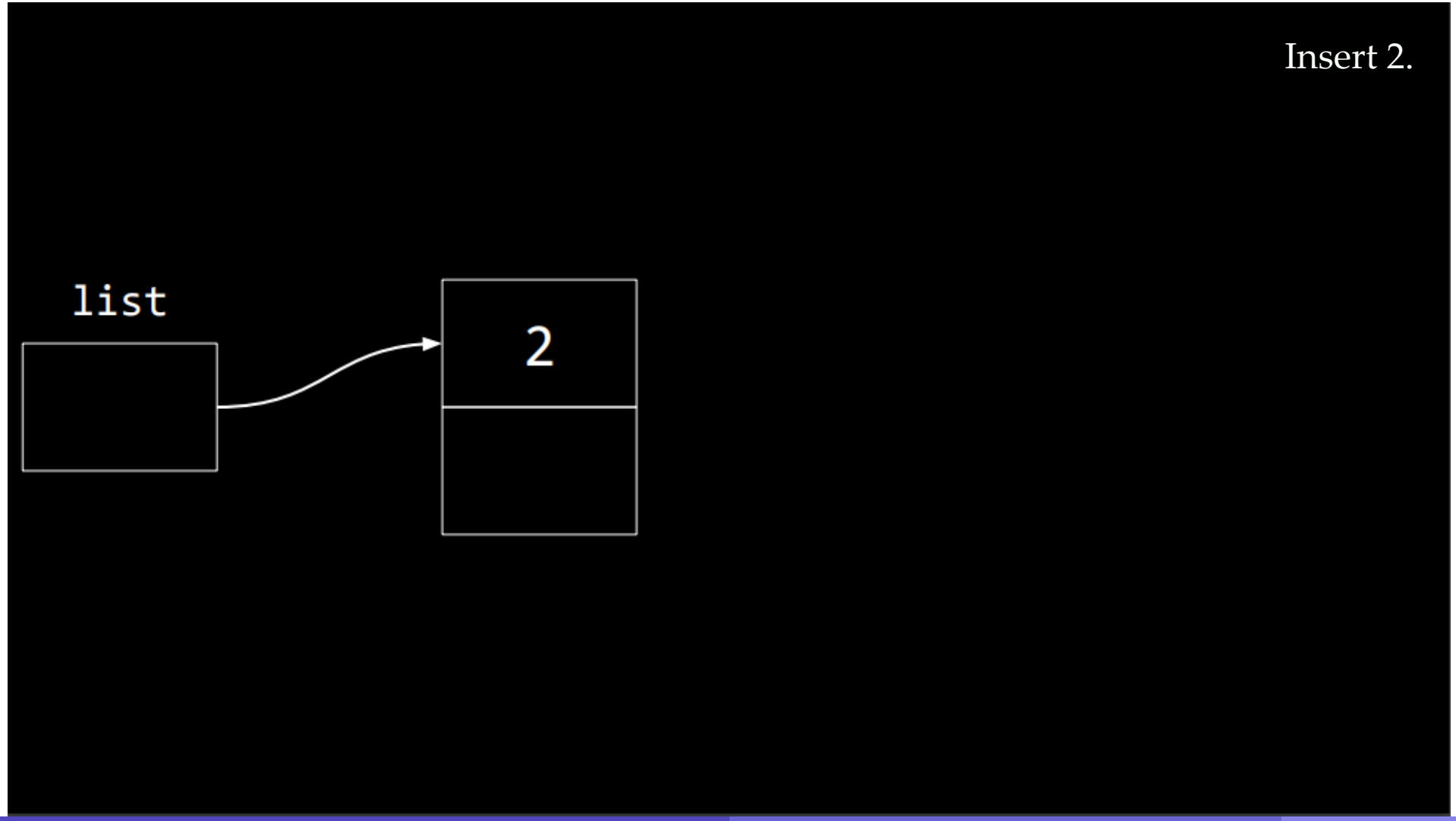
We append, prepend, or splice based on the situation.



Singly Linked List

Inserting a node (*sorted order*)

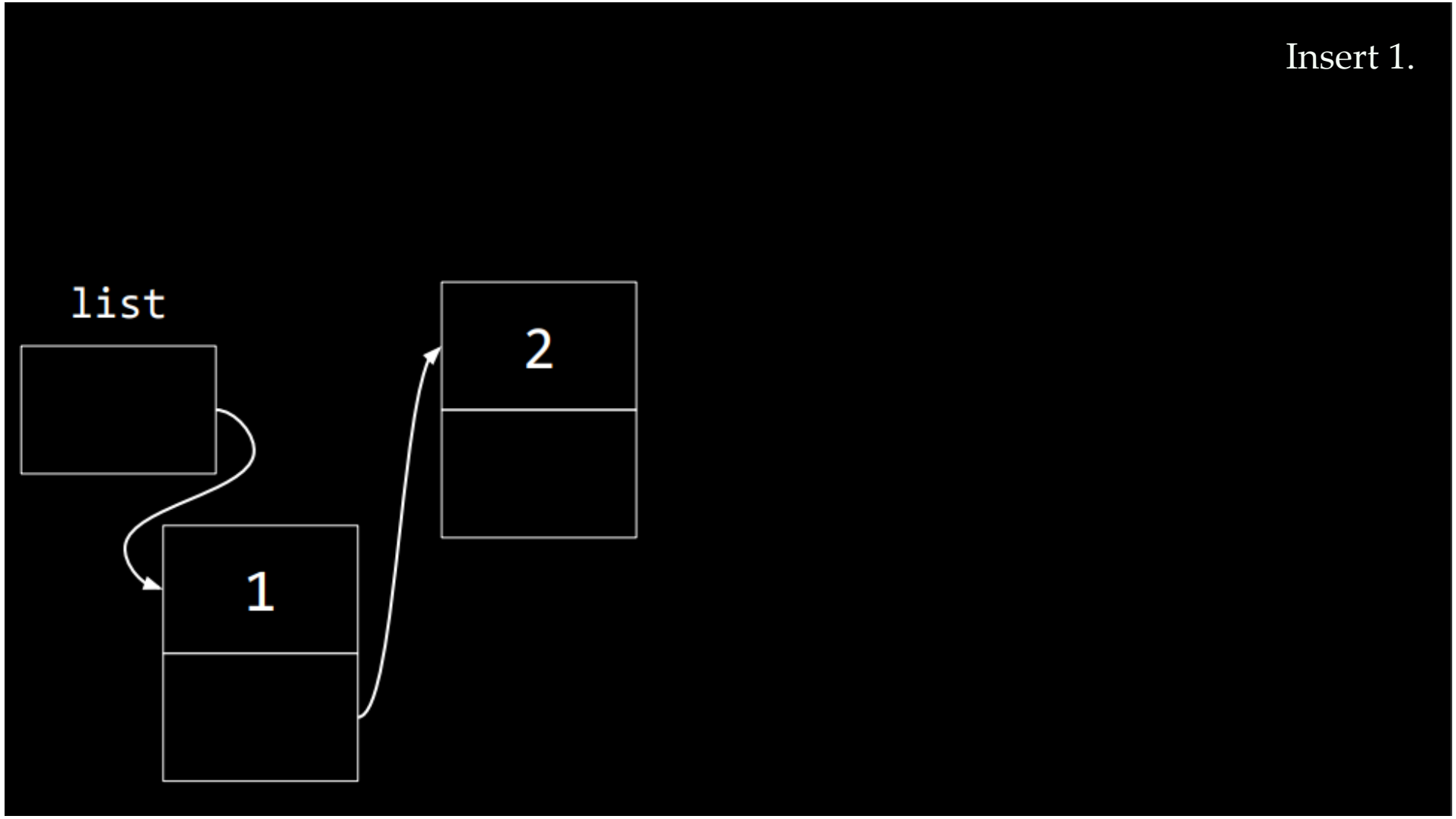
We append, prepend, or splice based on the situation.



Singly Linked List

Inserting a node (*sorted order*)

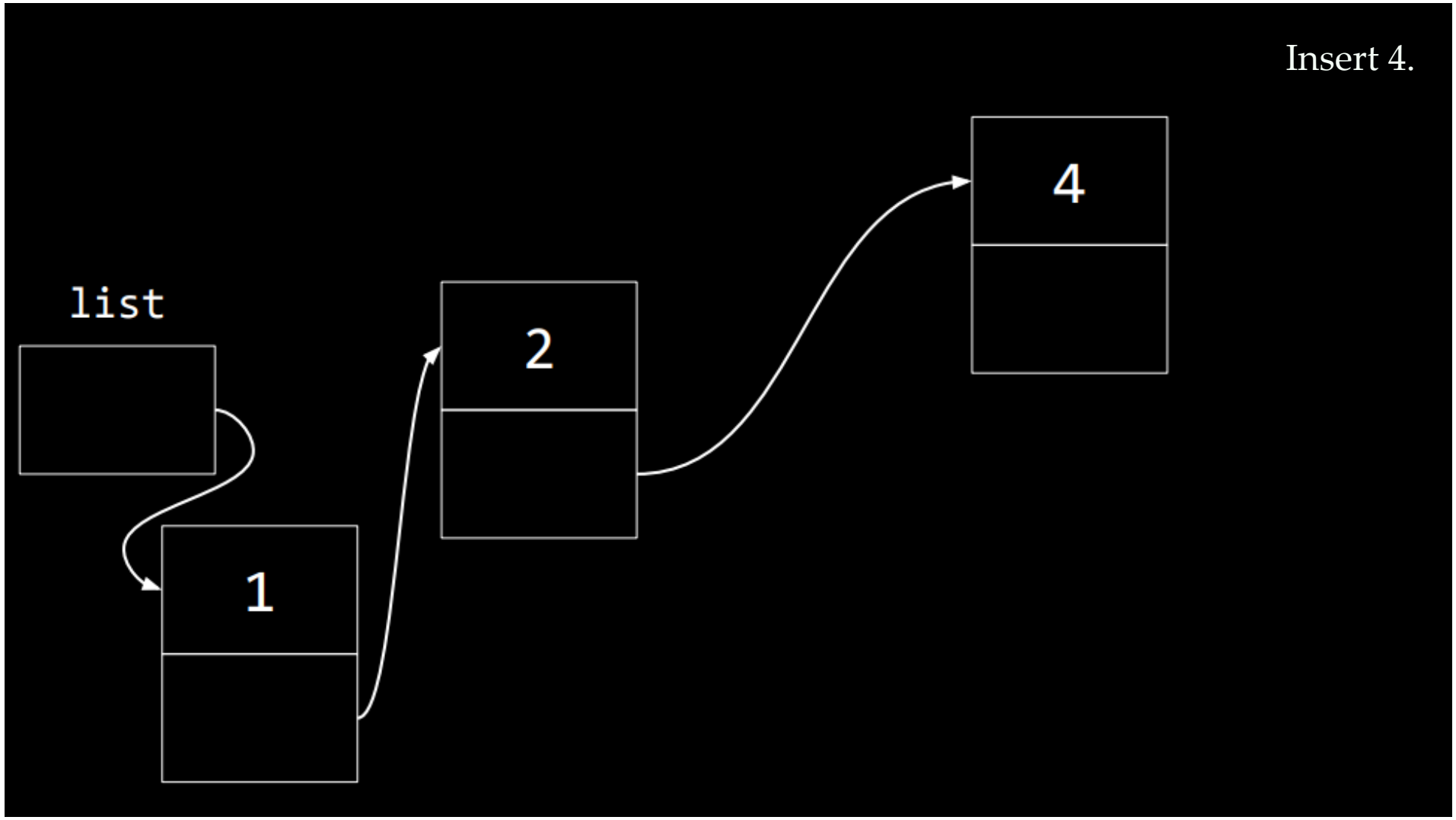
We append, prepend, or splice based on the situation.



Singly Linked List

Inserting a node (*sorted order*)

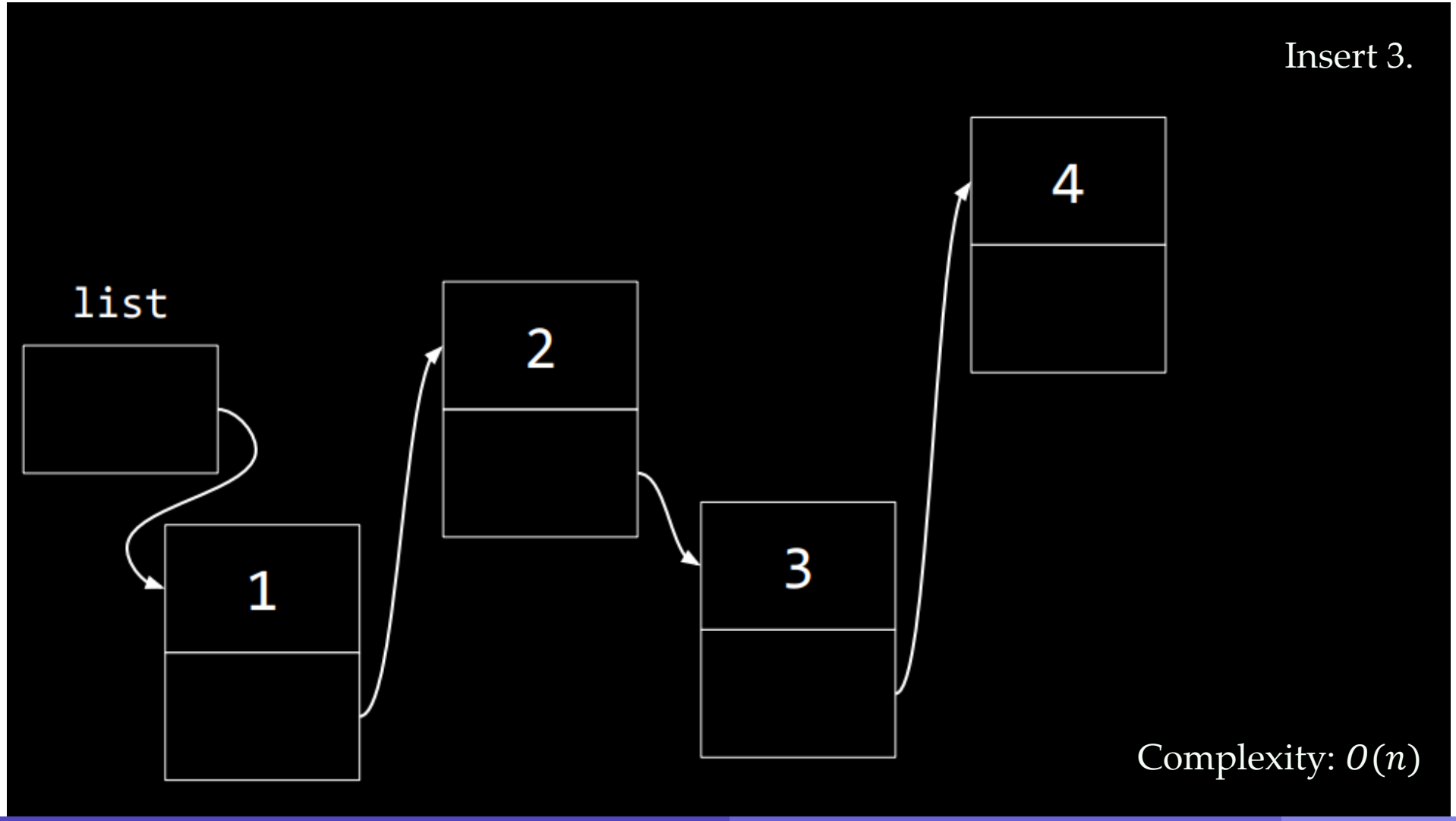
We append, prepend, or splice based on the situation.



Singly Linked List

Inserting a node (*sorted order*)

We append, prepend, or splice based on the situation.



Singly Linked List

Inserting a node (*sorted order*)

```
void insert_sorted(int element)
{
    node* n = (node*)malloc(sizeof(node));
    n->value = element;
    n->next = NULL;

    if(list == NULL)
    {
        list = n;
        return;
    }
    else if(n->value < list->value)
    {
        n->next = list;
        list = n;
    }
    else
    {
        for(node* curr = list; curr != NULL; curr = curr->next)
        {
            if(curr->next == NULL)
            {
                curr->next = n;
                break;
            }
            if(n->value < curr->next->value)
            {
                n->next = curr->next;
                curr->next = n;
                break;
            }
        }
    }
}
```

Prepending

Appending

Splicing

```
int main()
{
    int x;
    while(scanf("%d", &x) != EOF)
    {
        insert_sorted(x);
    }
    print_list();
    ...
    ...
    ...
```

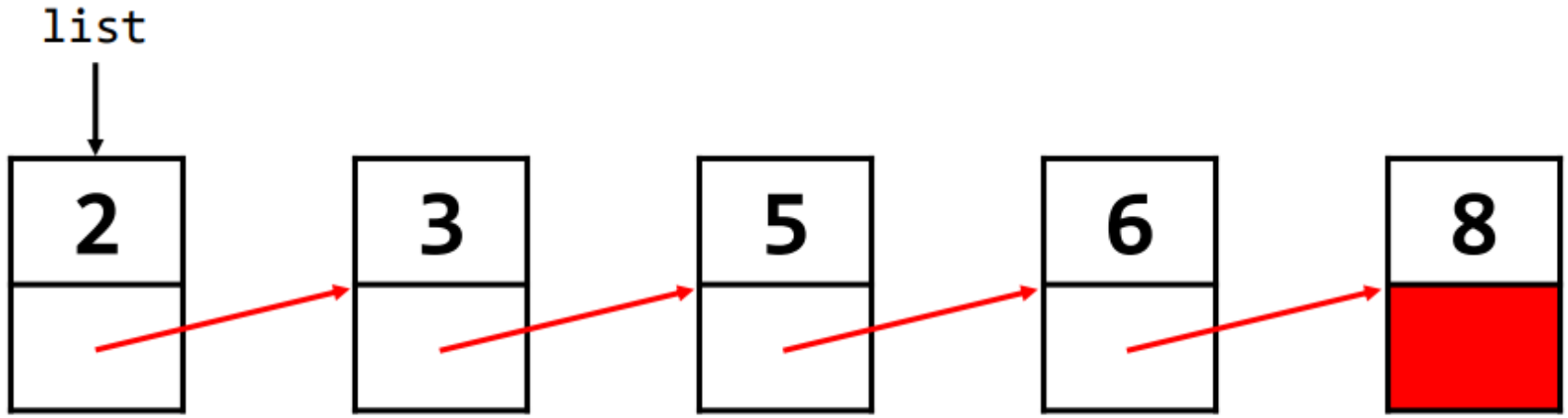
```
"D:\IUT Teaching\CSE4202 Structured Programming II Lab\Demo C Programs\singlylinkedlist.exe"
2 1 4 3
^Z
1 2 3 4

Process returned 0 (0x0)   execution time : 13.213 s
Press any key to continue.
```

Singly Linked List

Searching for an element

We search for the element by traversing through the entire linked list.

```
bool find(int element)
{
    for(node* trav = list; trav != NULL; trav = trav->next)
    {
        if(trav->value == element)
        {
            return true;
        }
    }
    return false;
}
```

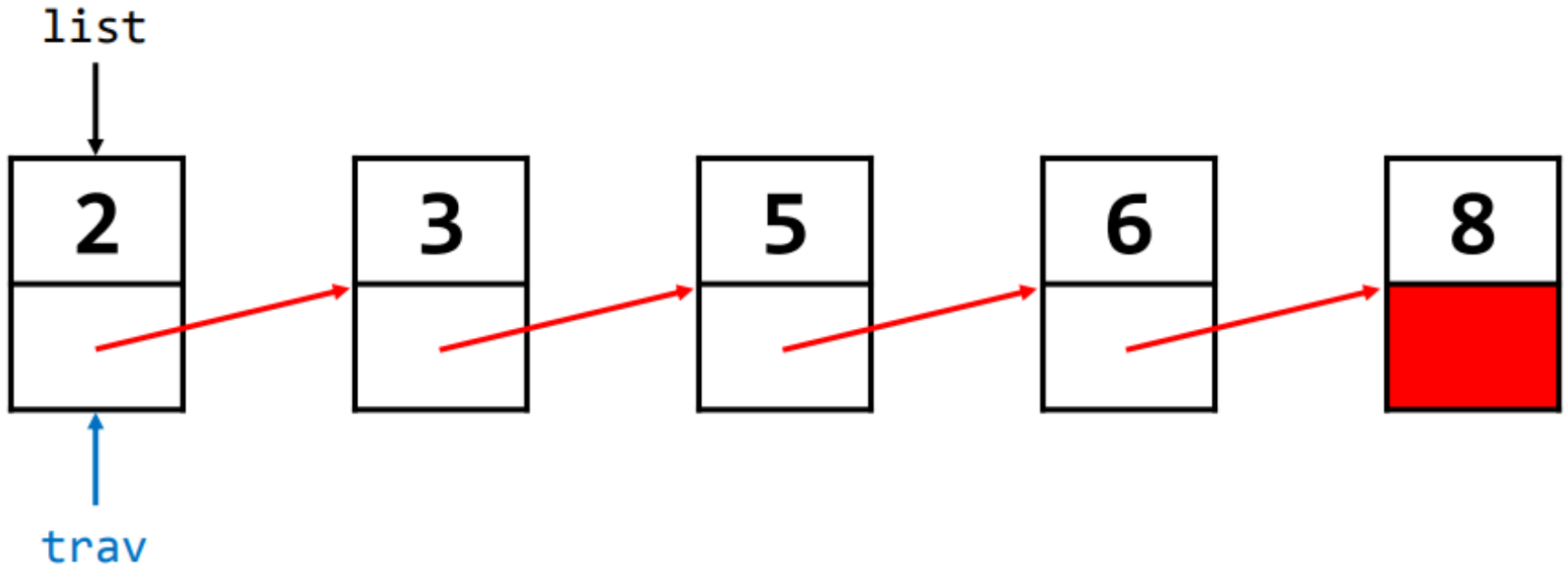


Singly Linked List

Searching for an element

We search for the element by traversing through the entire linked list.

```
bool find(int element)
{
    for(node* trav = list; trav != NULL; trav = trav->next)
    {
        if(trav->value == element)
        {
            return true;
        }
    }
    return false;
}
```

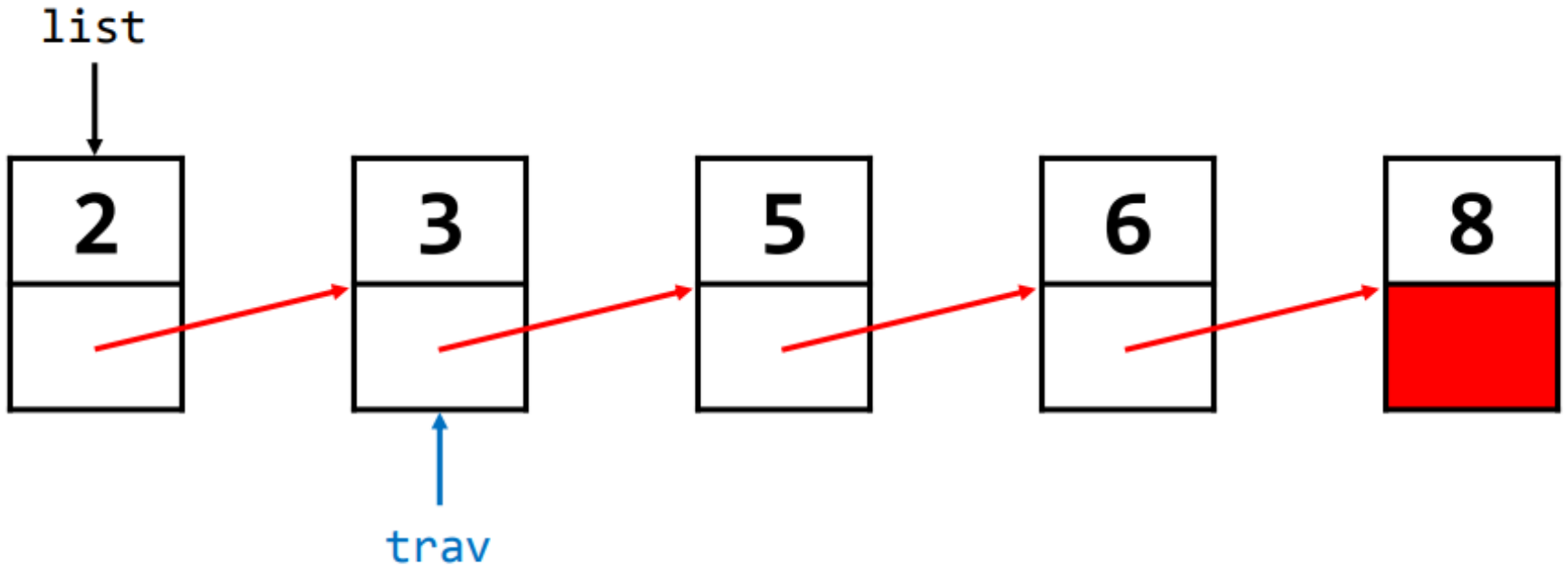


Singly Linked List

Searching for an element

We search for the element by traversing through the entire linked list.

```
bool find(int element)
{
    for(node* trav = list; trav != NULL; trav = trav->next)
    {
        if(trav->value == element)
        {
            return true;
        }
    }
    return false;
}
```

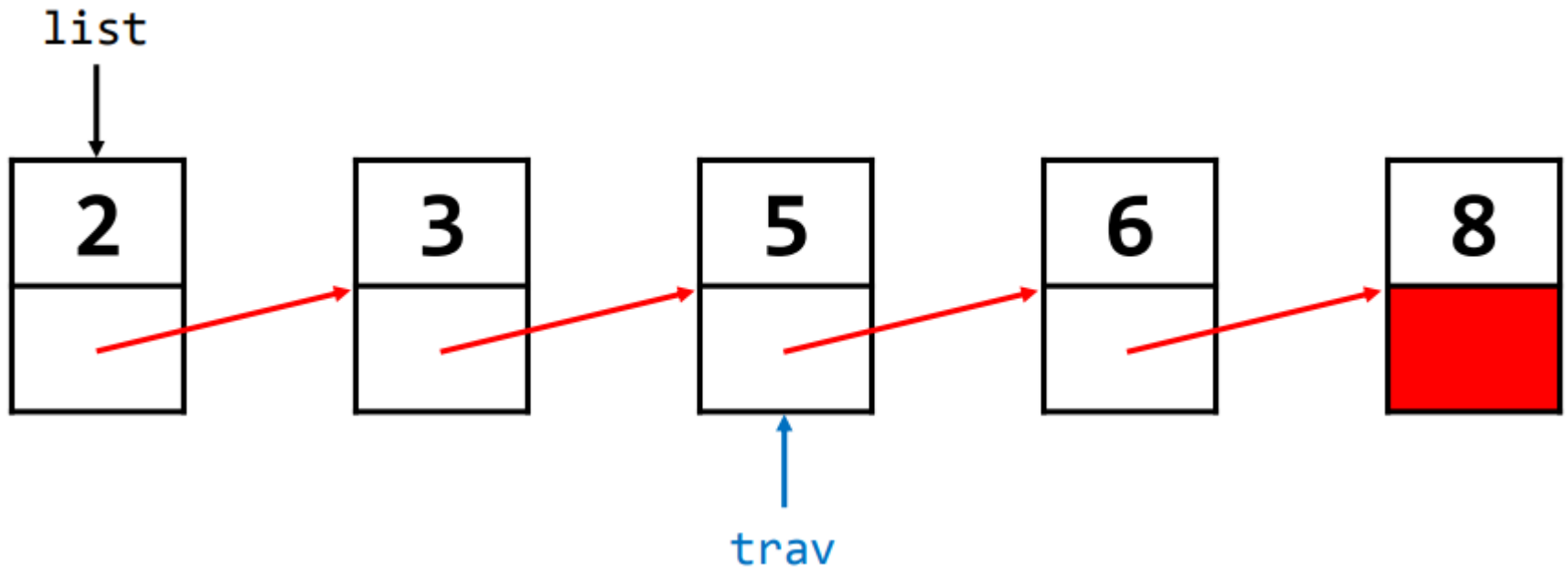


Singly Linked List

Searching for an element

We search for the element by traversing through the entire linked list.

```
bool find(int element)
{
    for(node* trav = list; trav != NULL; trav = trav->next)
    {
        if(trav->value == element)
        {
            return true;
        }
    }
    return false;
}
```



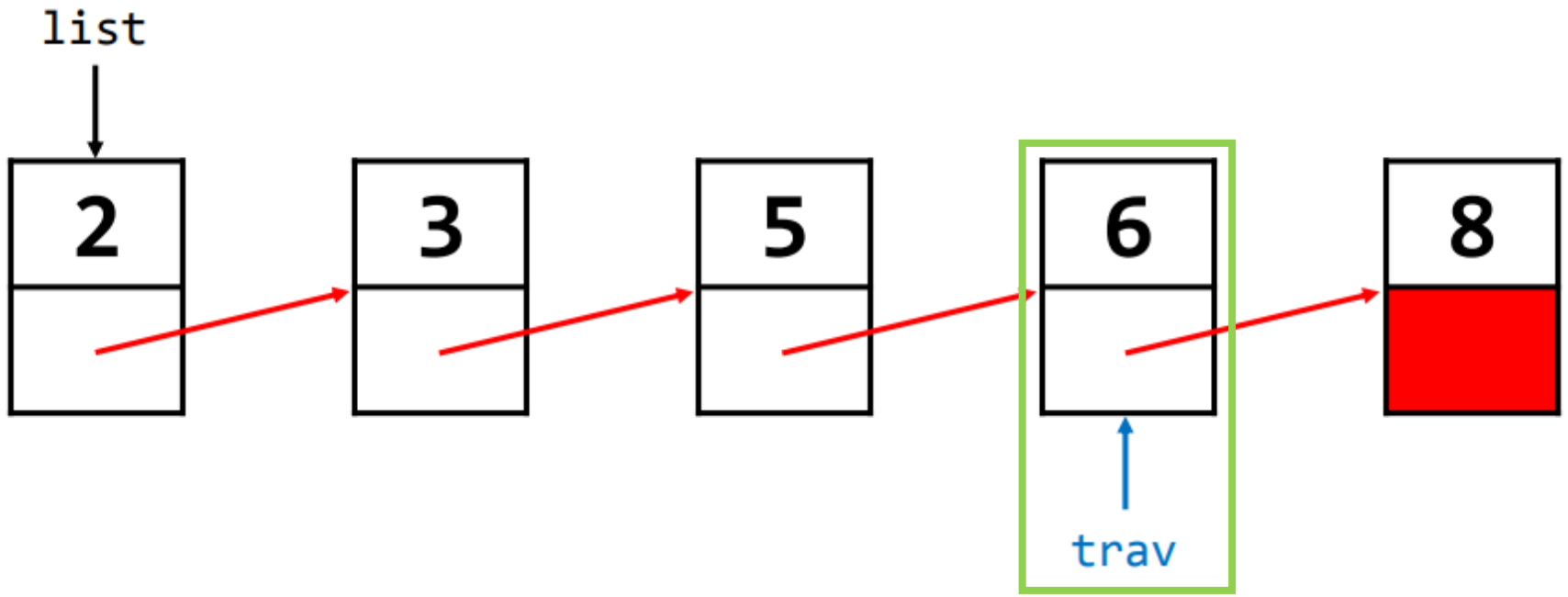
Singly Linked List

Searching for an element

We search for the element by traversing through the entire linked list.

```
bool find(int element)
{
    for(node* trav = list; trav != NULL; trav = trav->next)
    {
        if(trav->value == element)
        {
            return true;
        }
    }
    return false;
}
```

Complexity: $O(n)$



Singly Linked List

Inserting a node (*after an element*)

We search for the element, and insert the new node after it.

```
void insert_after(int element, int pred)
{
    node *n = (node*)malloc(sizeof(node));
    n->value = element;
    n->next = NULL;

    node* curr = list;
    while(curr->value != pred)
    {
        curr = curr->next;
    }
    n->next = curr->next;
    curr->next = n;
}
```

```
int main()
{
    int x;
    while(scanf("%d", &x) != EOF)
    {
        insert_sorted(x);
    }
    print_list();
    int k = 3, new_element = 13;
    insert_after(new_element, k);
    printf("%d has been inserted after %d.\n", new_element, k);
    print_list();
    ...
    ...
    ...
```

```
"D:\IUT Teaching\CSE4202 Structured Programming II Lab\Demo C Programs\singlylinkedlist.exe"
1 2 3 4
^Z
1 2 3 4
13 has been inserted after 3.
1 2 3 13 4

Process returned 0 (0x0)   execution time : 8.489 s
Press any key to continue.
```

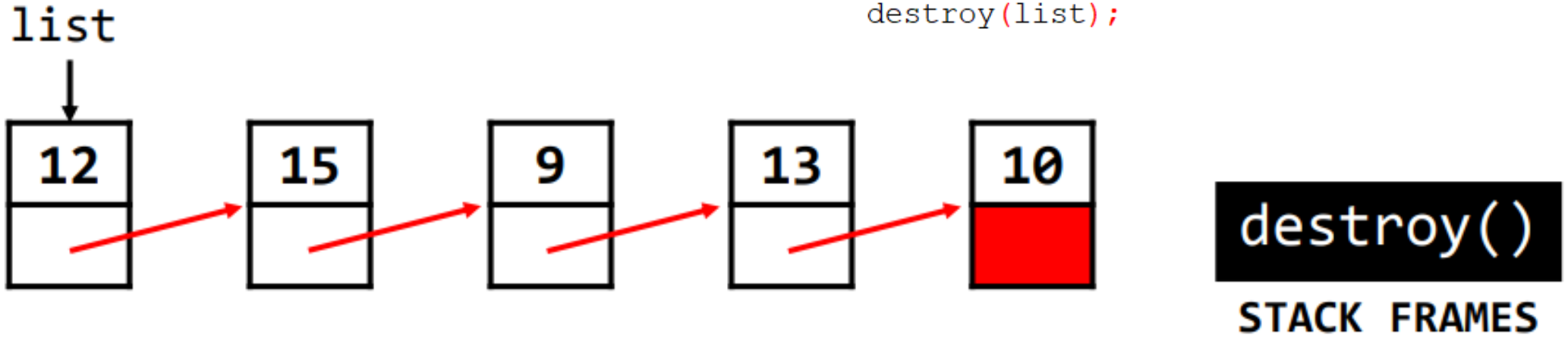
Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//          printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

```
int main()
{
    int x;
    while(scanf("%d", &x) != EOF)
    {
        insert_sorted(x);
    }
    print_list();
    ...
    ...
    ...
    destroy(list);
```

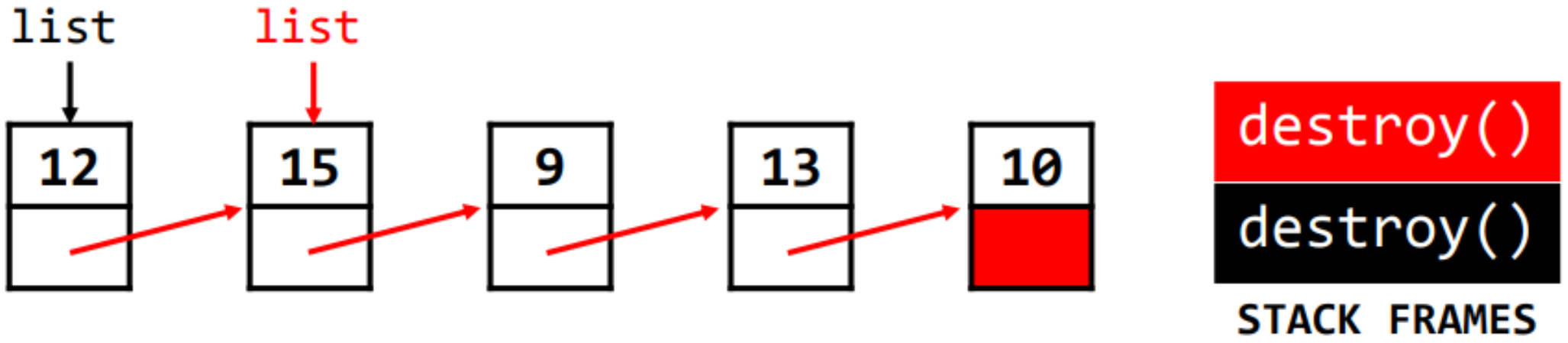


Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//        printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

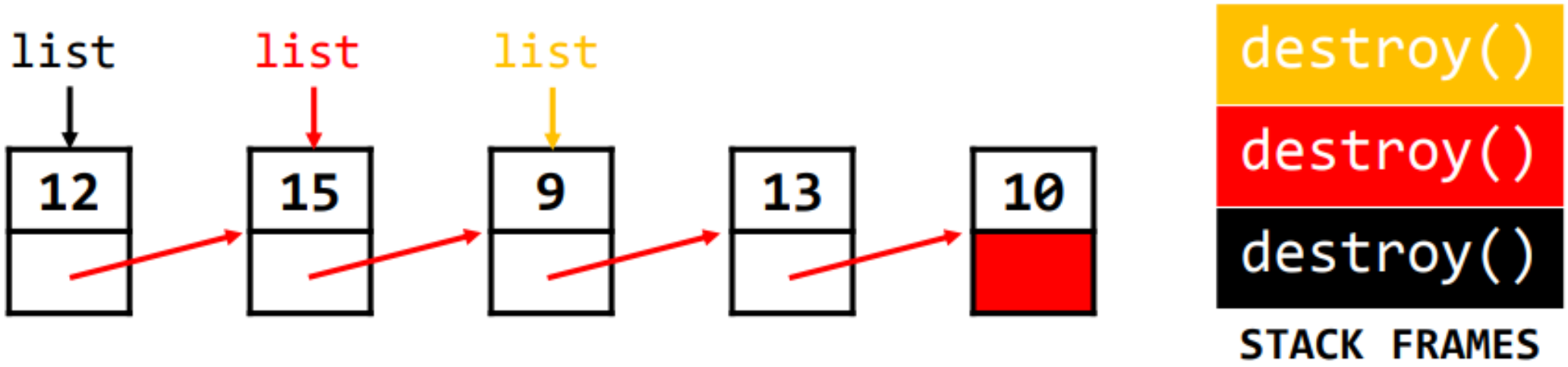


Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//        printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

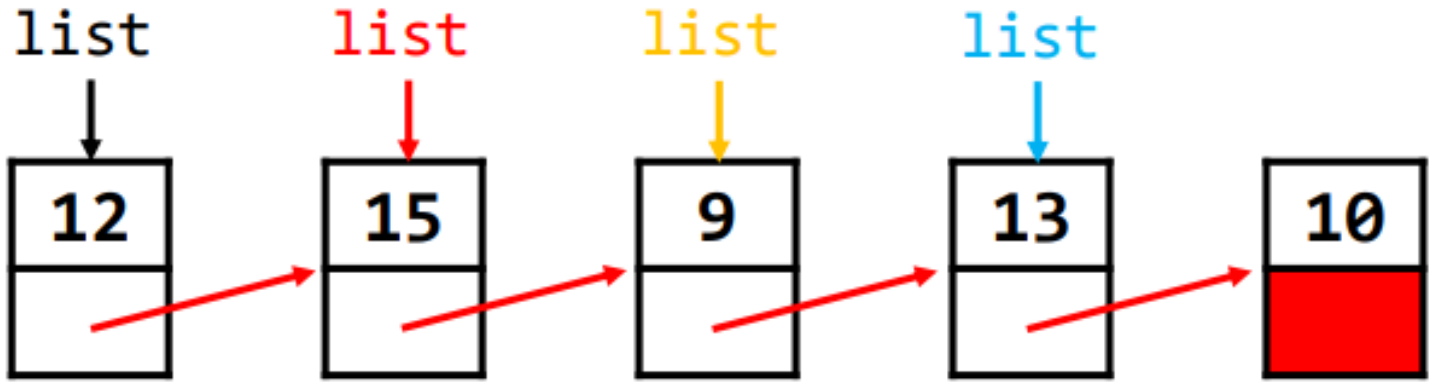


Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//          printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

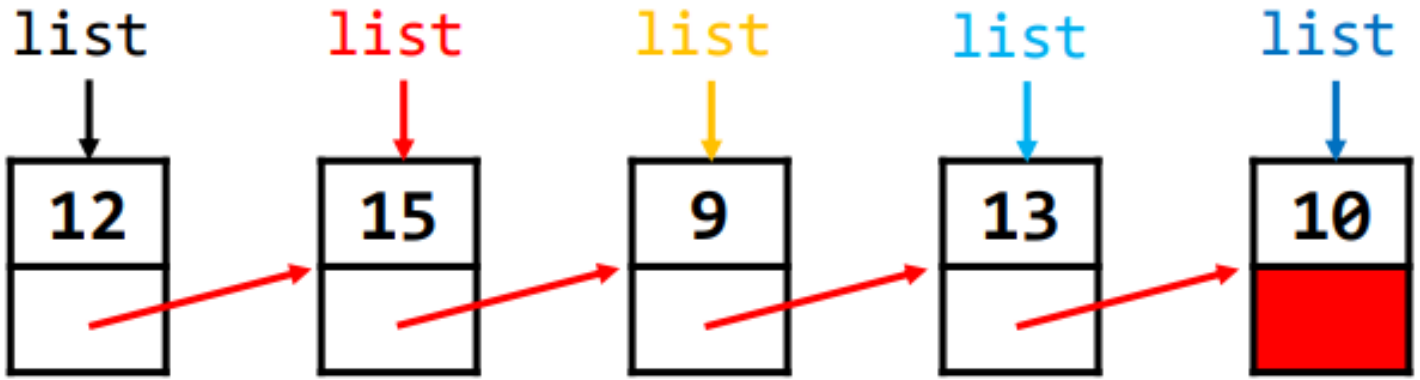


Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//          printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

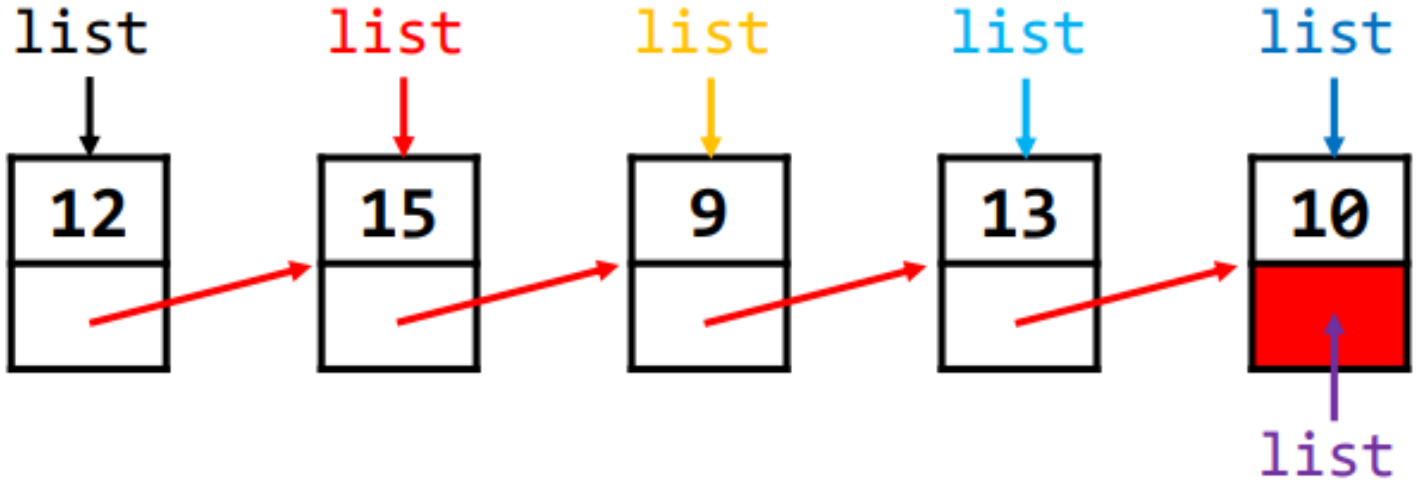


Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//          printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

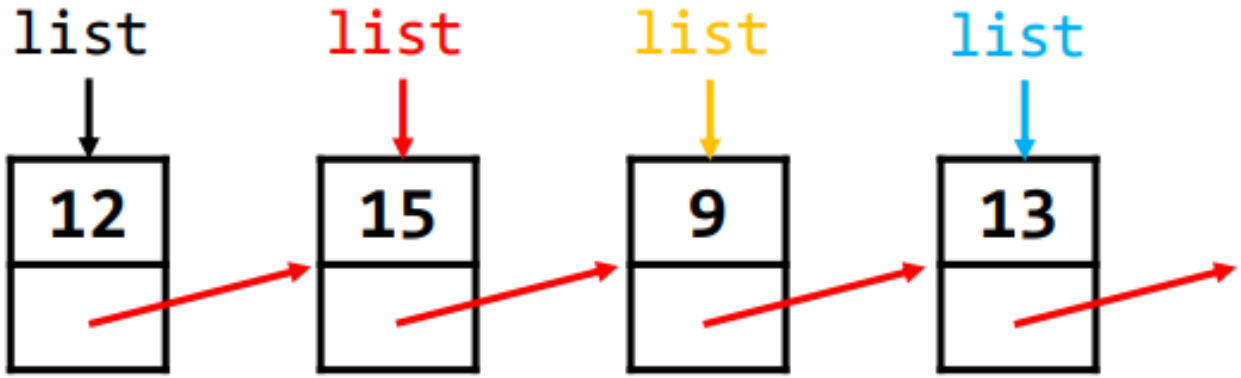


Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//          printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

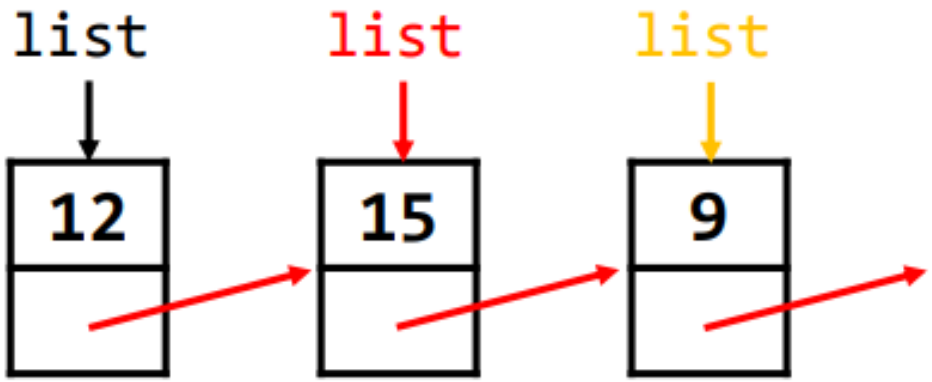


Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//        printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

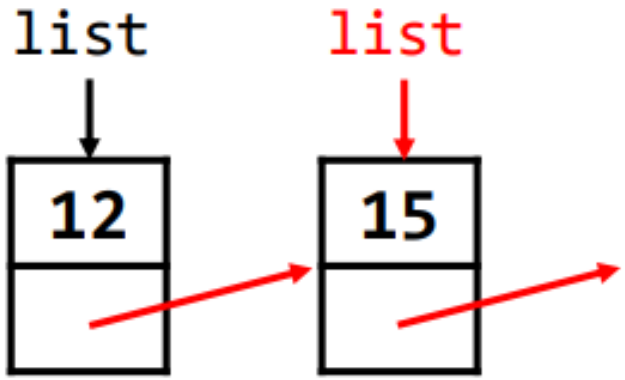


Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//        printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

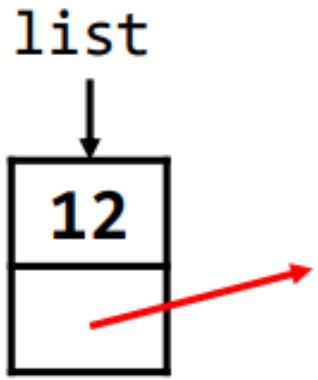


Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//        printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```



Singly Linked List

Deleting the entire list

We need to let go of the dynamically allocated memory slots to avoid memory leaks!

```
void destroy(node* n)
{
    if(n == NULL)
    {
//          printf("List is empty!\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

Poof!

STACK FRAMES

Singly Linked List

Deleting from the front of the list

To remove the first node from the linked list.

```
void delete_front()
{
    if(list == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    node* temp;
    temp = list;
    list = list->next;
    free(temp);
}
```

```
int main()
{
    int x;
    while(scanf("%d", &x) != EOF)
    {
        insert_sorted(x);
    }
    print_list();
    delete_front();
    printf("The first element has been deleted.\n");
    print_list();
    ...
    ...
    ...
```

```
Select "D:\IUT Teaching\CSE4202 Structured Programming II Lab\Demo C Programs\singlylinkedlist.exe"
1 2 3 4
^Z
1 2 3 4
The first element has been deleted.
2 3 4

Process returned 0 (0x0)   execution time : 3.528 s
Press any key to continue.
```

Singly Linked List

Deleting from the back of the list

To remove the first node from the linked list.

```
void delete_back()
{
    if(list == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    node* curr = list;
    node* prev = NULL;
    while(curr->next != NULL)
    {
        prev = curr;
        curr = curr->next;
    }
    if(prev != NULL)
    {
        prev->next = curr->next;
    }
    free(curr);
}
```

```
int main()
{
    int x;
    while(scanf("%d", &x) != EOF)
    {
        insert_sorted(x);
    }
    print_list();
    delete_back();
    printf("The last element has been deleted.\n");
    print_list();
    ...
    ...
    ...
```

```
"D:\IUT Teaching\CSE4202 Structured Programming II Lab\Demo C Programs\singlylinkedlist.exe"
1 2 3 4
^Z
1 2 3 4
The last element has been deleted.
1 2 3

Process returned 0 (0x0)   execution time : 2.870 s
Press any key to continue.
```

Singly Linked List

Deleting a particular element

Search for the element, then delete it.

```
void delete_node(int element)
{
    if(list == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    node* curr = list;
    node* prev = NULL;
    while(curr->value != element)
    {
        prev = curr;
        curr = curr->next;
    }
    if(prev != NULL)
    {
        prev->next = curr->next;
    }
    free(curr);
}
```

```
int main()
{
    int x;
    while(scanf("%d", &x) != EOF)
    {
        insert_sorted(x);
    }
    print_list();
    int k = 3;
    delete_node(k);
    printf("%d has been deleted.\n", k);
    print_list();
    ...
    ...
    ...
```

```
"D:\IUT Teaching\CSE4202 Structured Programming II Lab\Demo C Programs\singlylinkedlist.exe"
1 2 3 4
^Z
1 2 3 4
3 has been deleted.
1 2 4

Process returned 0 (0x0)   execution time : 4.595 s
Press any key to continue.
```

Linked List Variants

Types and nomenclature

There are a plethora of other approaches to implementing a linked list, each with their own advantages and disadvantages —

- Doubly Linked List
- Circular Linked List

} Your assignment for this lab.

- Multiply Linked List
- Linked List with Hash Linking

Now that you understand the ***basic building blocks***, it will be easy to conceptualize other data structures like Binary Tree, Binary Search Tree, AVL Tree, Trie, Hash Table, Stack, Queue, Heap, etc. (in the ***next semester***, insha'Allah)