Herzlich Willkommen bei Inf-Einf-B Block 5.

Heute geht es um Datenstrukturen.

Dies ist in unserem Kurs die *vorletzte* Woche, in der wir die Sprache C behandeln.

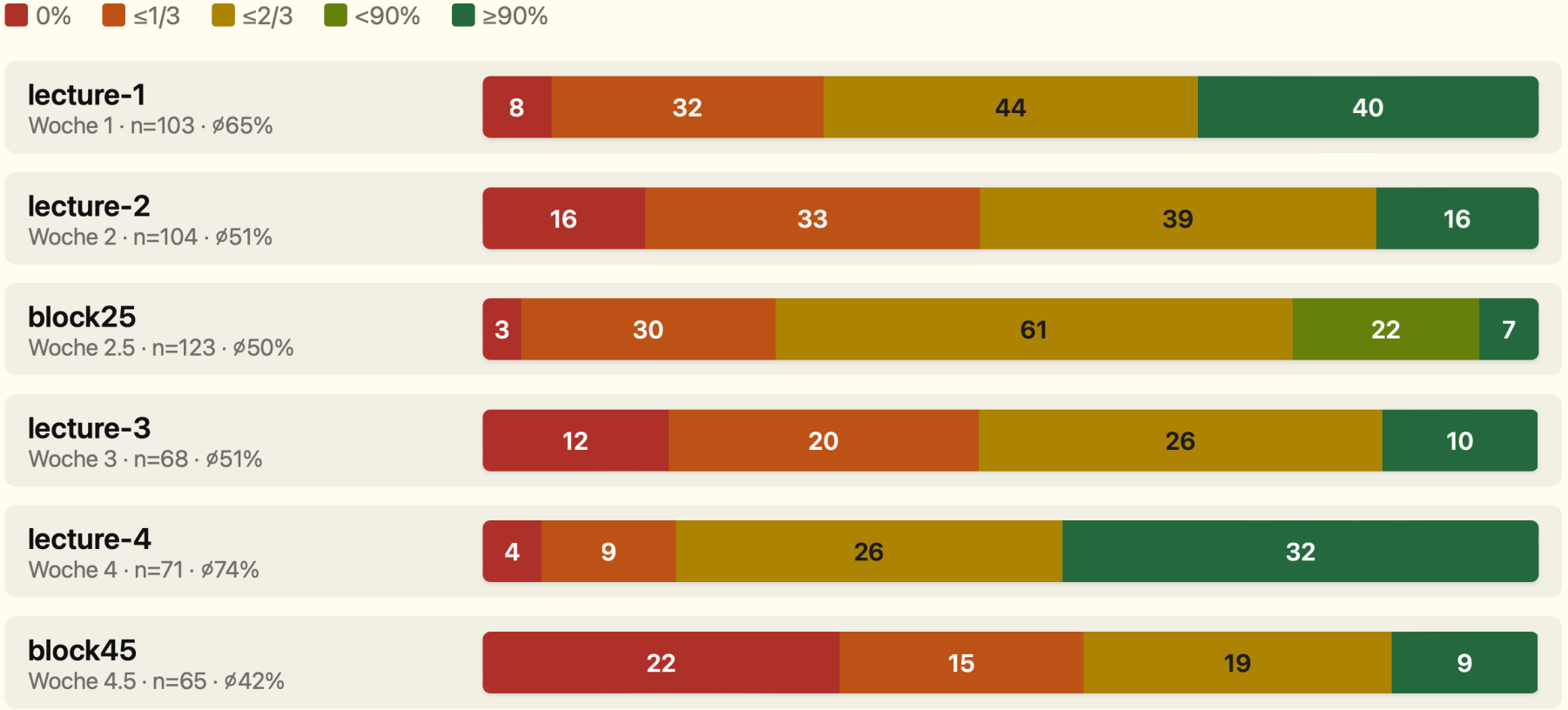
Zwei Drittel der Inhalte haben wir fast geschafft!

Abweichend zu CS50 behandeln wir einige Inhalte *nur* in der Vorlesung und nicht in den Shorts, der Section und der Übung: doppelt verkettete Listen, Queues und Hash Tabellen.

Wir machen heute wieder eine Pause (25 min).

Dies ist Inf-Einf-B.

Kurs-Statistiken



Hochgeladene Übungsaufgaben

Woche 0 Scratch	Woche 1 C	Woche 2 Arrays	Woche 3 Algor.	Woche 4 Memory
88	130	115	78	80+

Wiederholung

```
// Wo liegen die Daten bei diesen Variablen?

int main(void)
{
  int x[100];
  int *y = malloc(100 * sizeof(int));
  char *s = "Hallo";
  char t[] = "Hallo";
  …
}
```

Wiederholung

```
// Wo liegen die Daten bei diesen Variablen?

int main(void)
{
  int x[100];
  int *y = malloc(100 * sizeof(int));
  char *s = "Hallo";
  char t[] = "Hallo";
  ...
}
```

x[100] → Stack (automatisch freigegeben)

*y = malloc(...) → Heap (manuell free nötig)

*s = "Hallo" → String-Literal im Read-Only-Bereich, Pointer s auf Stack

t[] = "Hallo" → Stack

PROBLEM 1

Was gibt das Programm aus und warum?

```c
#include <stdio.h>

void mystery(int *a, int *b)
{
    if (*a > *b) {
        int tmp = *a;
        *a = *b;
        *b = tmp;
    }
}

int main(void)
{
    int x = 3, y = 7;
    mystery(&x, &y);
    printf("%d %d\n", x, y);

    int p = 9, q = 2;
    mystery(&p, &q);
    printf("%d %d\n", p, q);
}
```

PROBLEM 1

```c
#include <stdio.h>

void mystery(int *a, int *b)
{
    if (*a > *b) {
        int tmp = *a;
        *a = *b;
        *b = tmp;
    }
}

int main(void)
{
    int x = 3, y = 7;
    mystery(&x, &y);
    printf("%d %d\n", x, y);

    int p = 9, q = 2;
    mystery(&p, &q);
    printf("%d %d\n", p, q);
}
```

Was gibt das Programm aus und warum?

3 7
2 9

Erster Aufruf: 3 > 7 ist false
→ kein Swap → "3 7"

Zweiter Aufruf: 9 > 2 ist true
→ Swap → "2 9"

Die Funktion tauscht, wenn das erste größer ist.

PROBLEM 2

```
void verdopple(int *p, int n)
{
    for (int i = 0; i < n; i++)
        p[i] *= 2;
}
```

Welche Aufrufe sind korrekt?

```
int a[3] = {1, 2, 3};
int *b = malloc(3 * sizeof(int));
int c = 7;

verdopple(a, 3);       // ?
verdopple(b, 3);       // ?
verdopple(&c, 1);      // ?
verdopple(&a[0], 3);   // ?
verdopple(a[0], 3);    // ?
verdopple(*b, 3);      // ?
```

PROBLEM 2

```
void verdopple(int *p, int n)
{
    for (int i = 0; i < n; i++)
        p[i] *= 2;
}
```

Welche Aufrufe sind korrekt?

```
int a[3] = {1, 2, 3};
int *b = malloc(3 * sizeof(int));
int c = 7;

verdopple(a, 3);       // ?  ✓ — Array zerfällt zu Pointer
verdopple(b, 3);       // ?  ✓ — b ist bereits int*
verdopple(&c, 1);      // ?  ✓ — Adresse einer einzelnen int
verdopple(&a[0], 3);   // ?  ✓ — Explizit: Adresse des ersten Elements
verdopple(a[0], 3);    // ?  x — a[0] ist int (Wert 1), nicht int*
verdopple(*b, 3);      // ?  x — *b dereferenziert, ist int, nicht int*
```

Datenstrukturen

Abstrakte Datentypen

Queues
(Warteschlangen)

FIFO

enqueue

dequeue

```
const int CAPACITY = 50;

typedef struct
{
    person people[CAPACITY];
    int size;
} queue;
```

Stacks
(Stapel)

LIFO

push

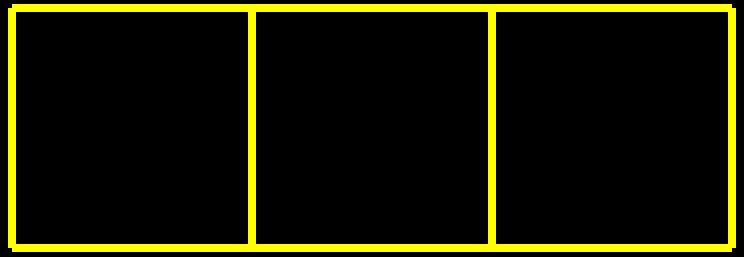
pop

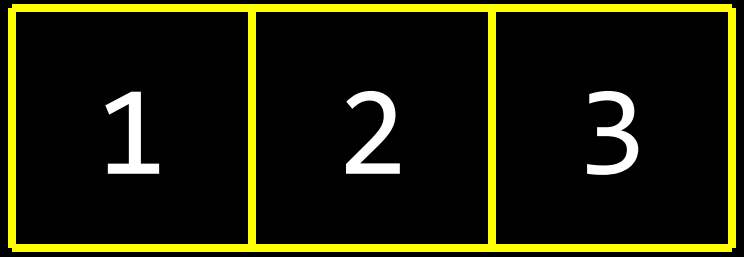
```
const int CAPACITY = 50;

typedef struct
{
    person people[CAPACITY];
    int size;
} stack;
```

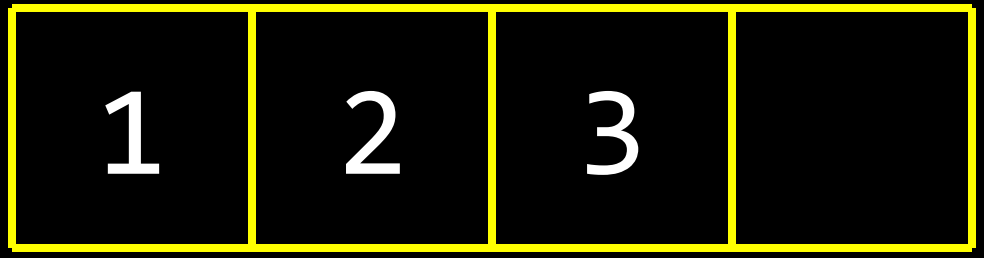
Implementierung von push und pop

im Short-Video zu Stacks

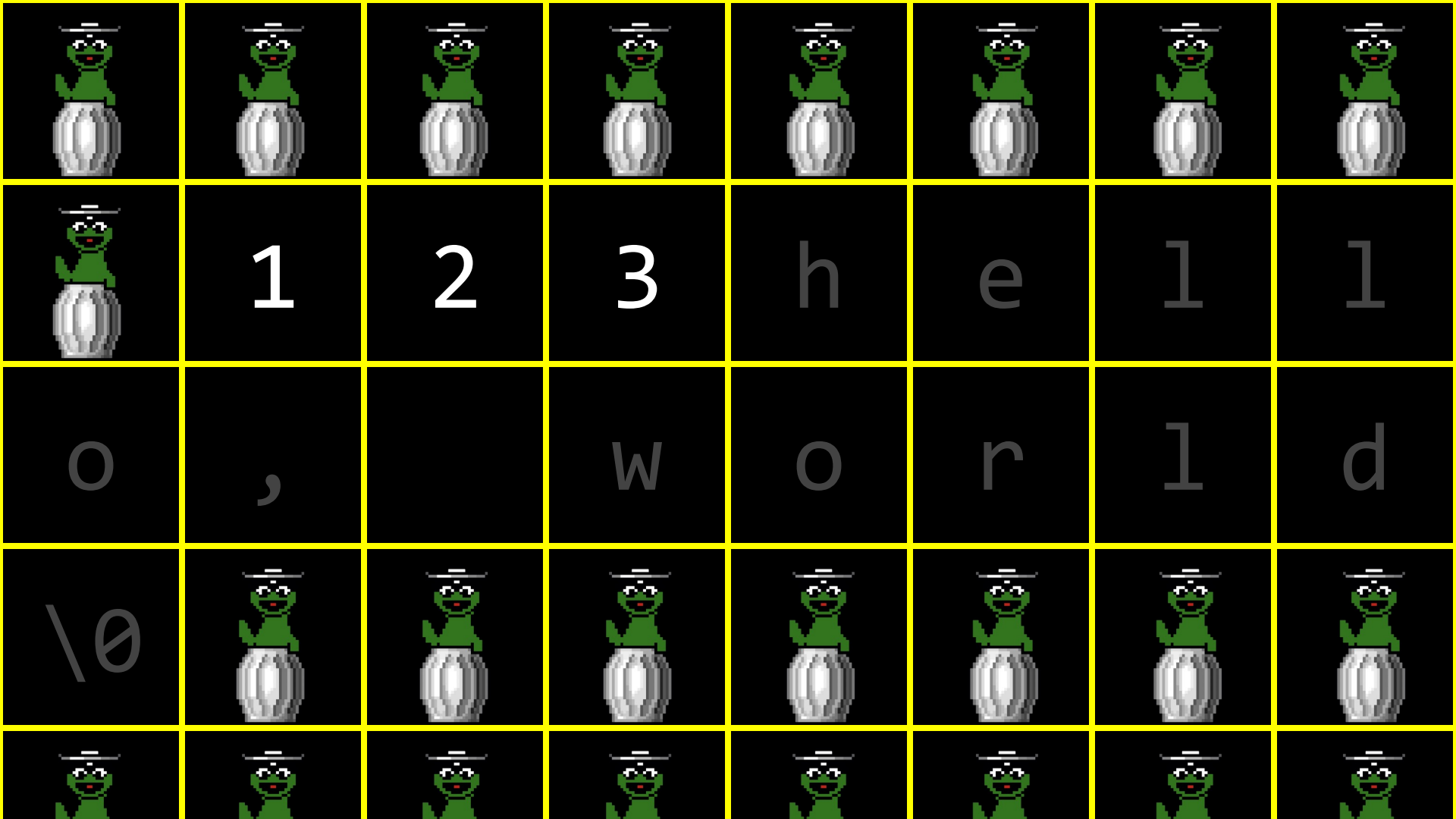
Arrays




1
2
3


1
2
3

1 2 3


1 2 3 h e l l
o , w o r l d
\0

1	2	3

1	2	3

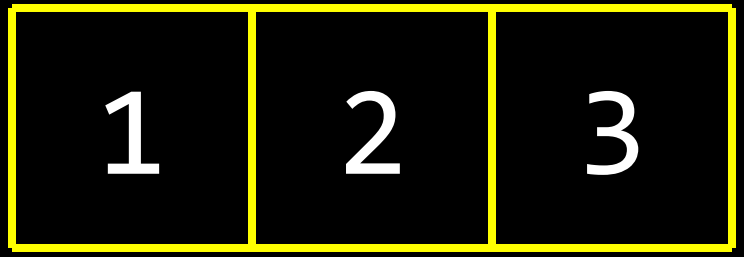
1			

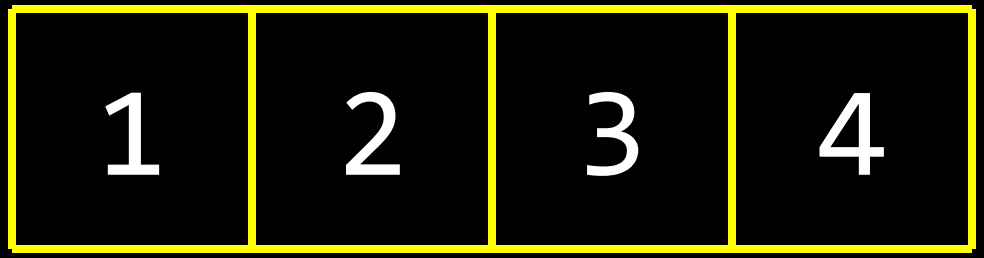
1	2	3

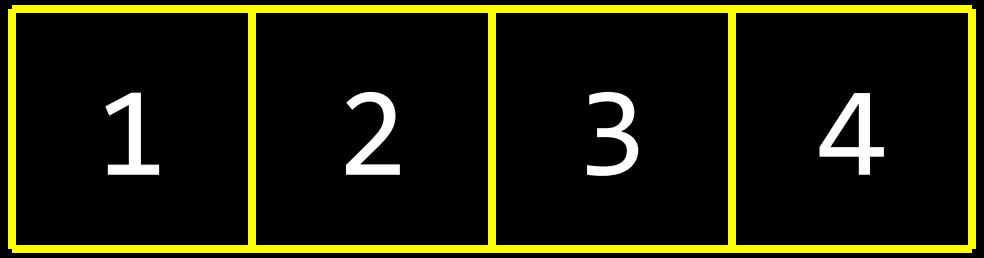
1	2		

1	2	3

1	2	3	


1
2
3


1
2
3
4


1
2
3
4

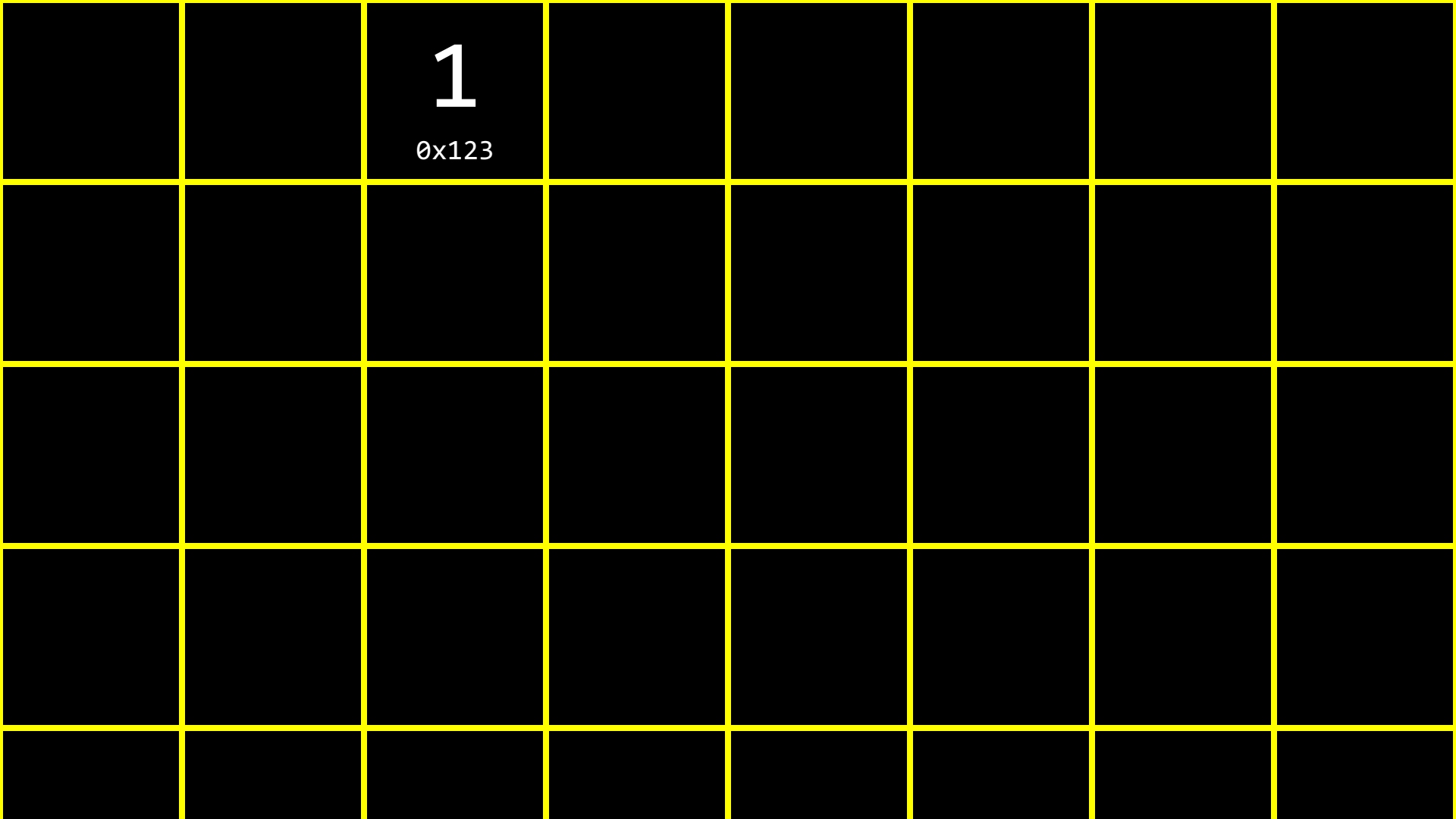
Strukturen
structs

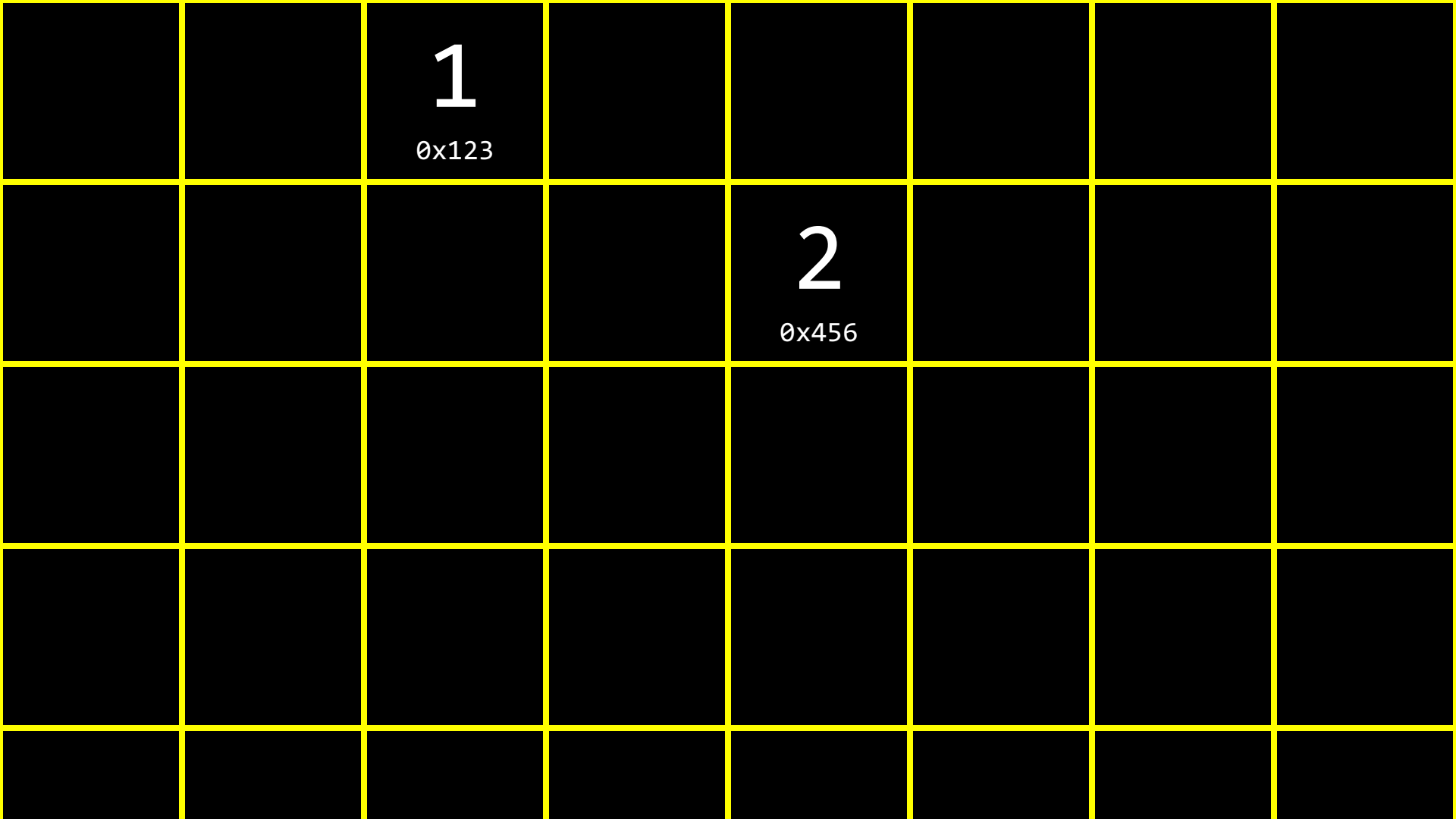
```
struct
.
*
```

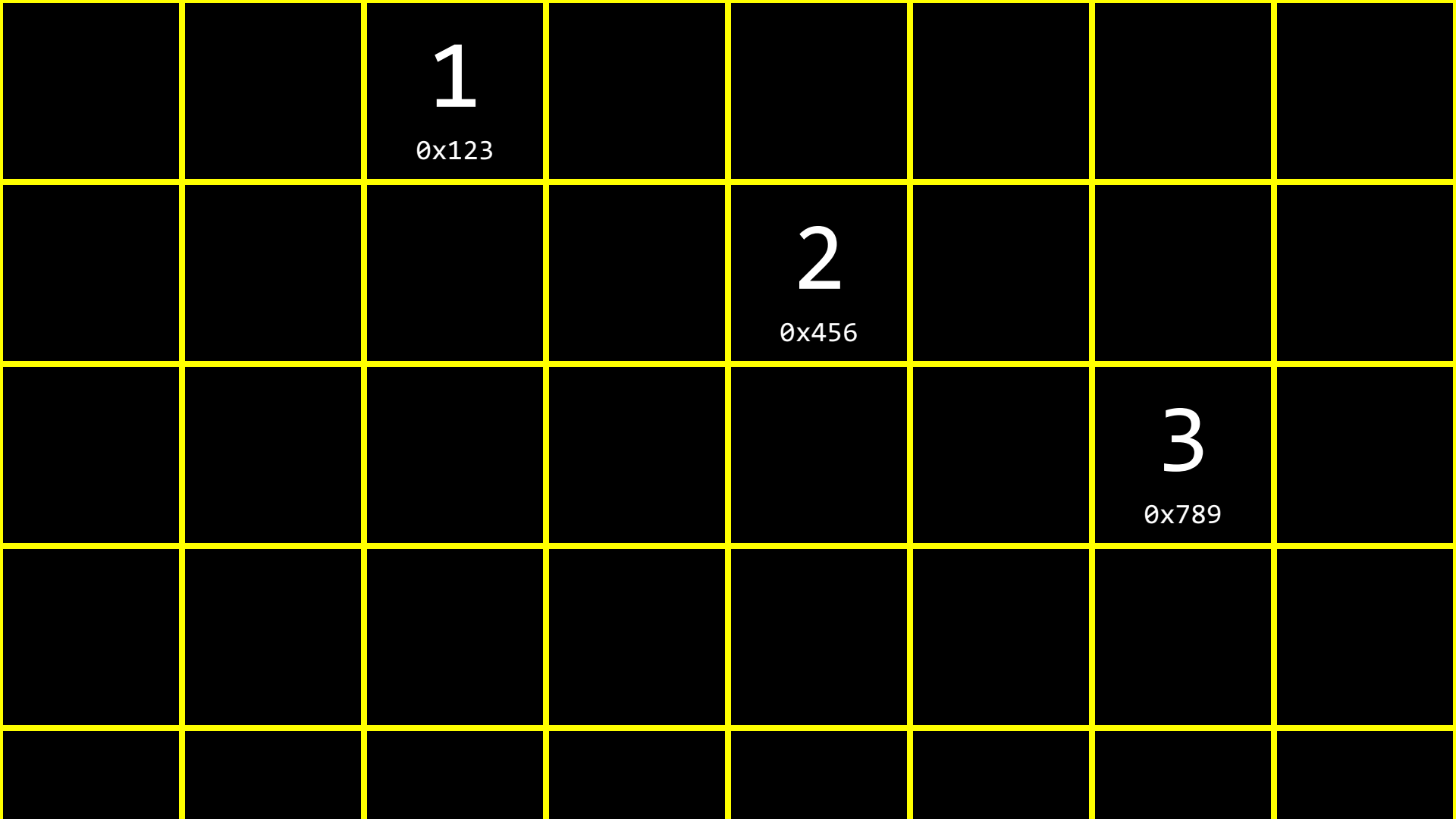
struct

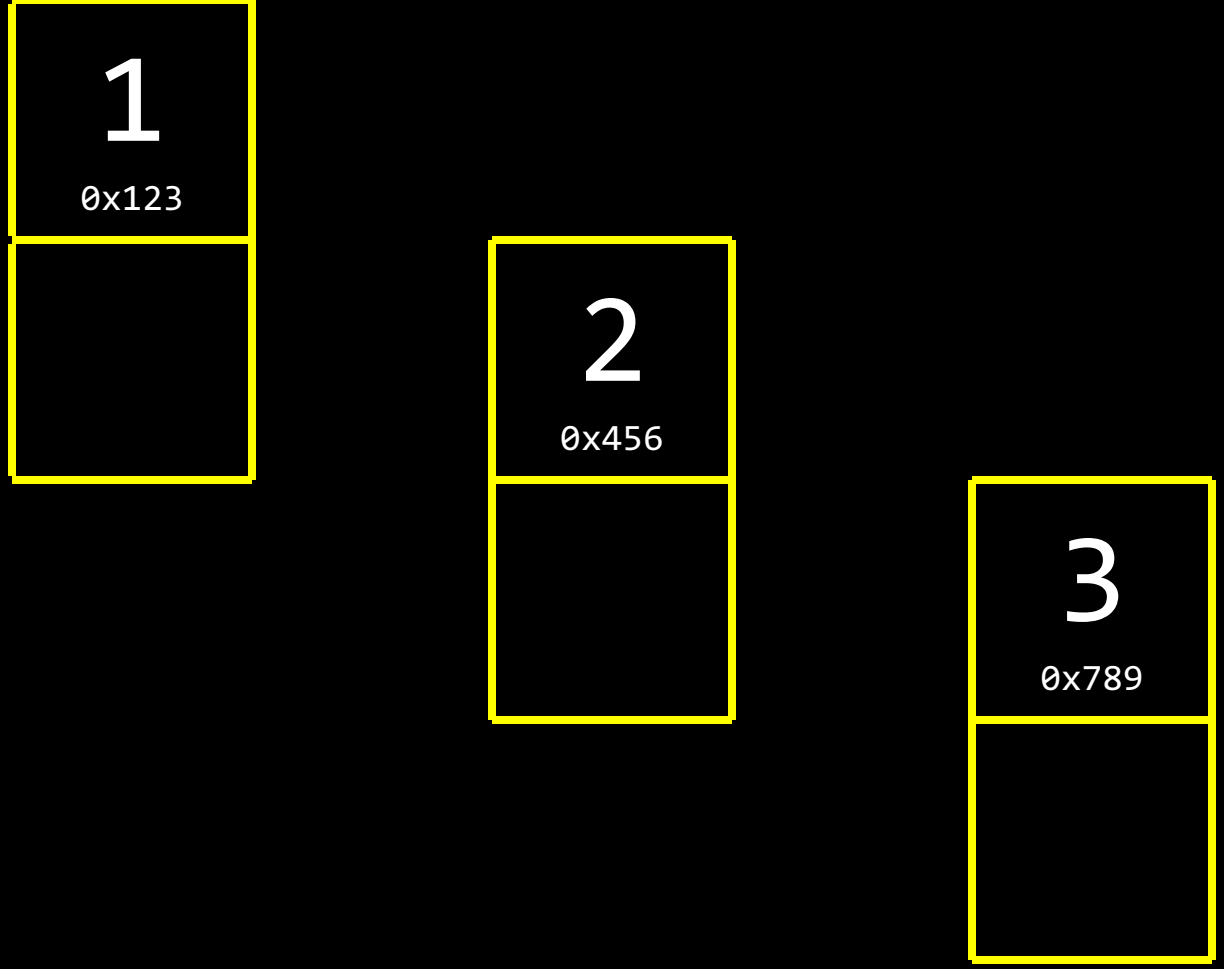
->

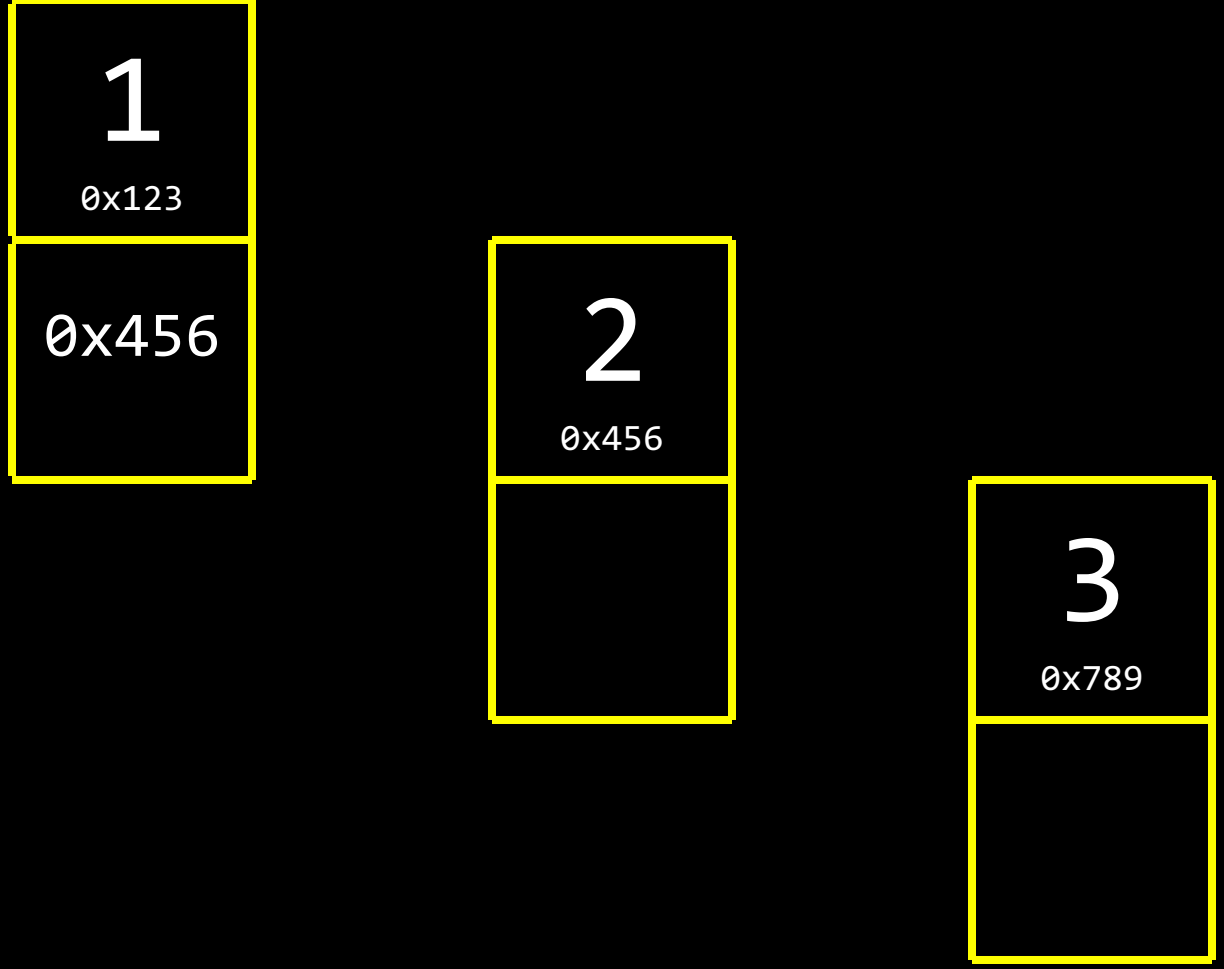
Verkettete Listen
(Linked Lists)

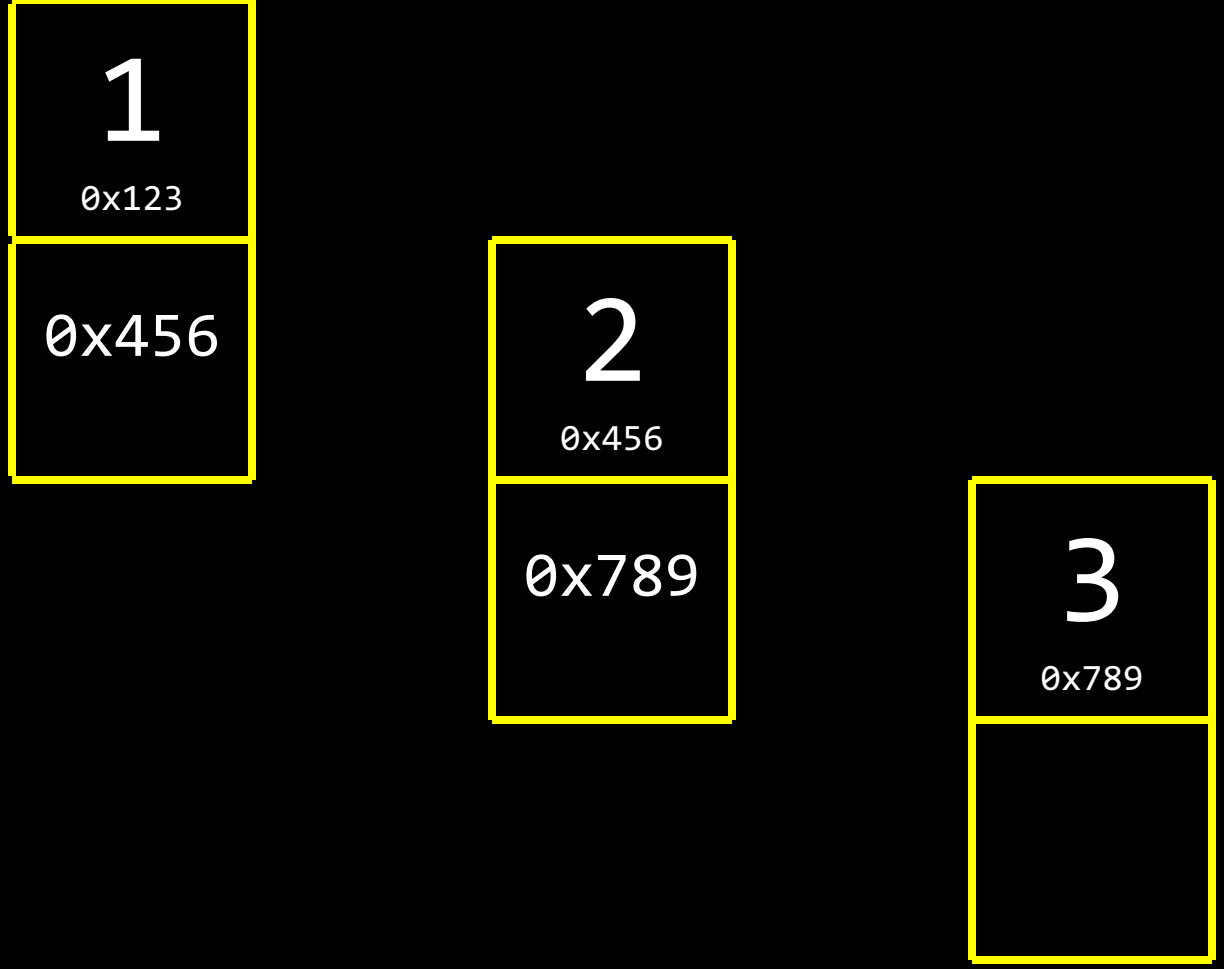

1
0x123

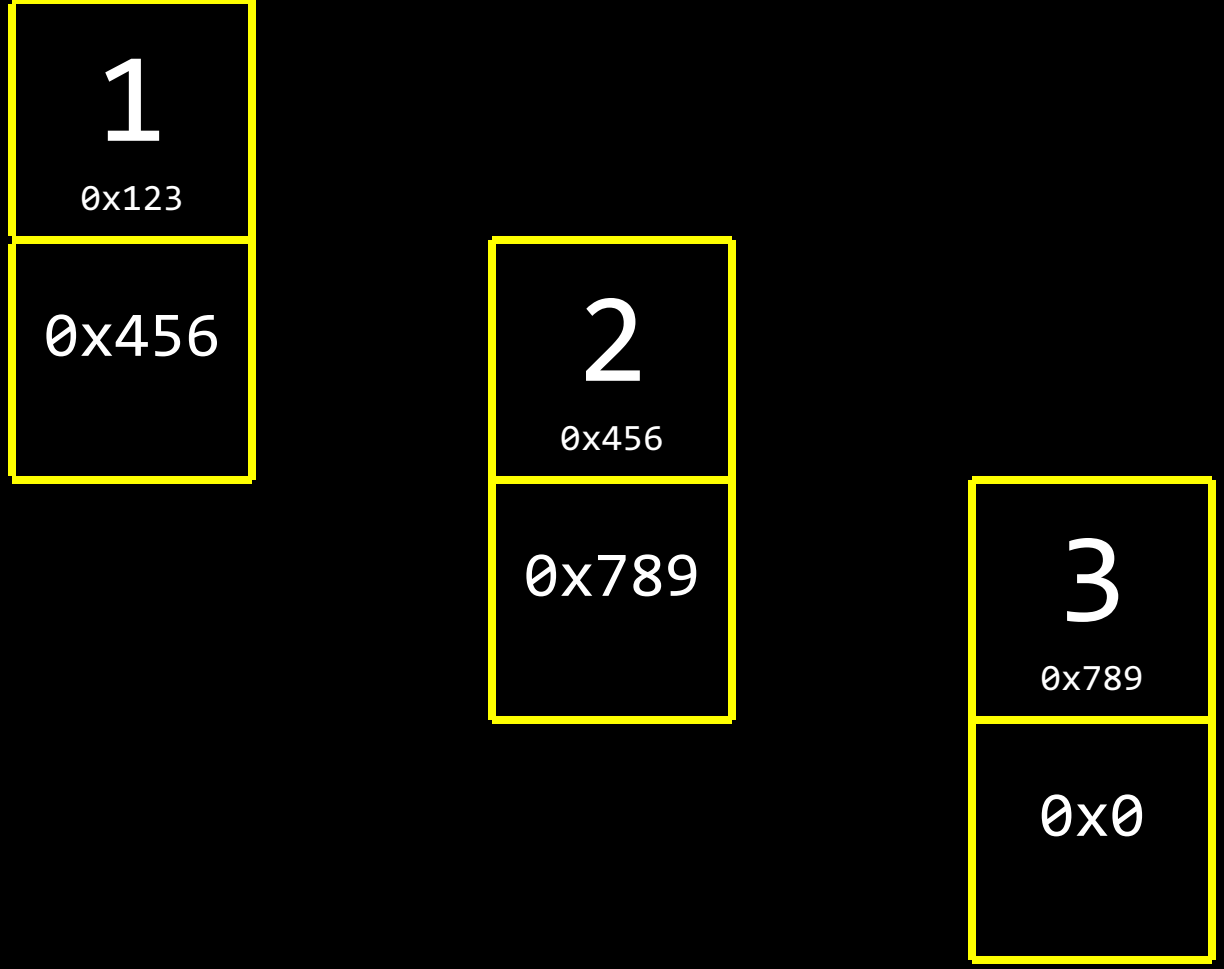

1
0x123
2
0x456

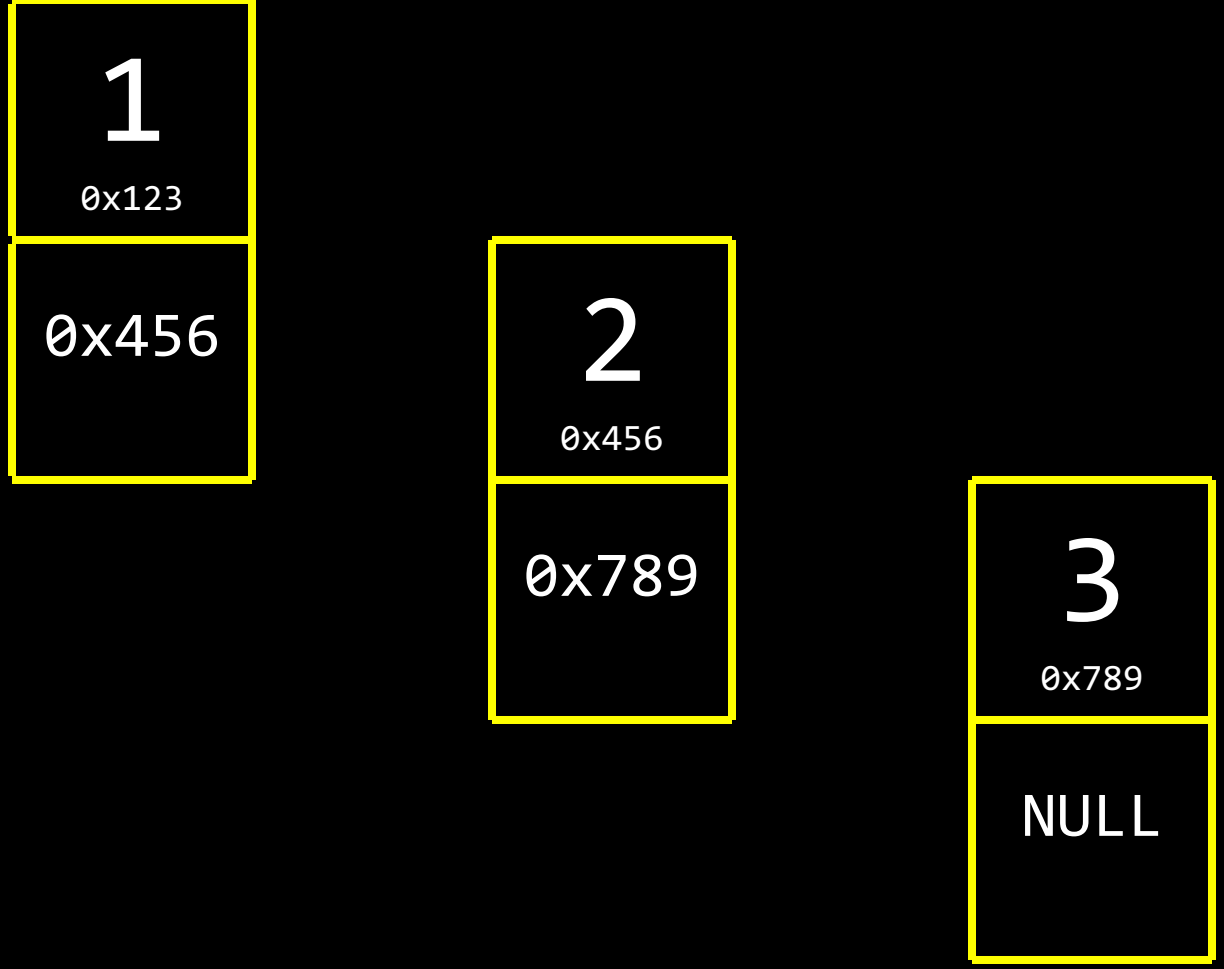

1
0x123
2
0x456
3
0x789

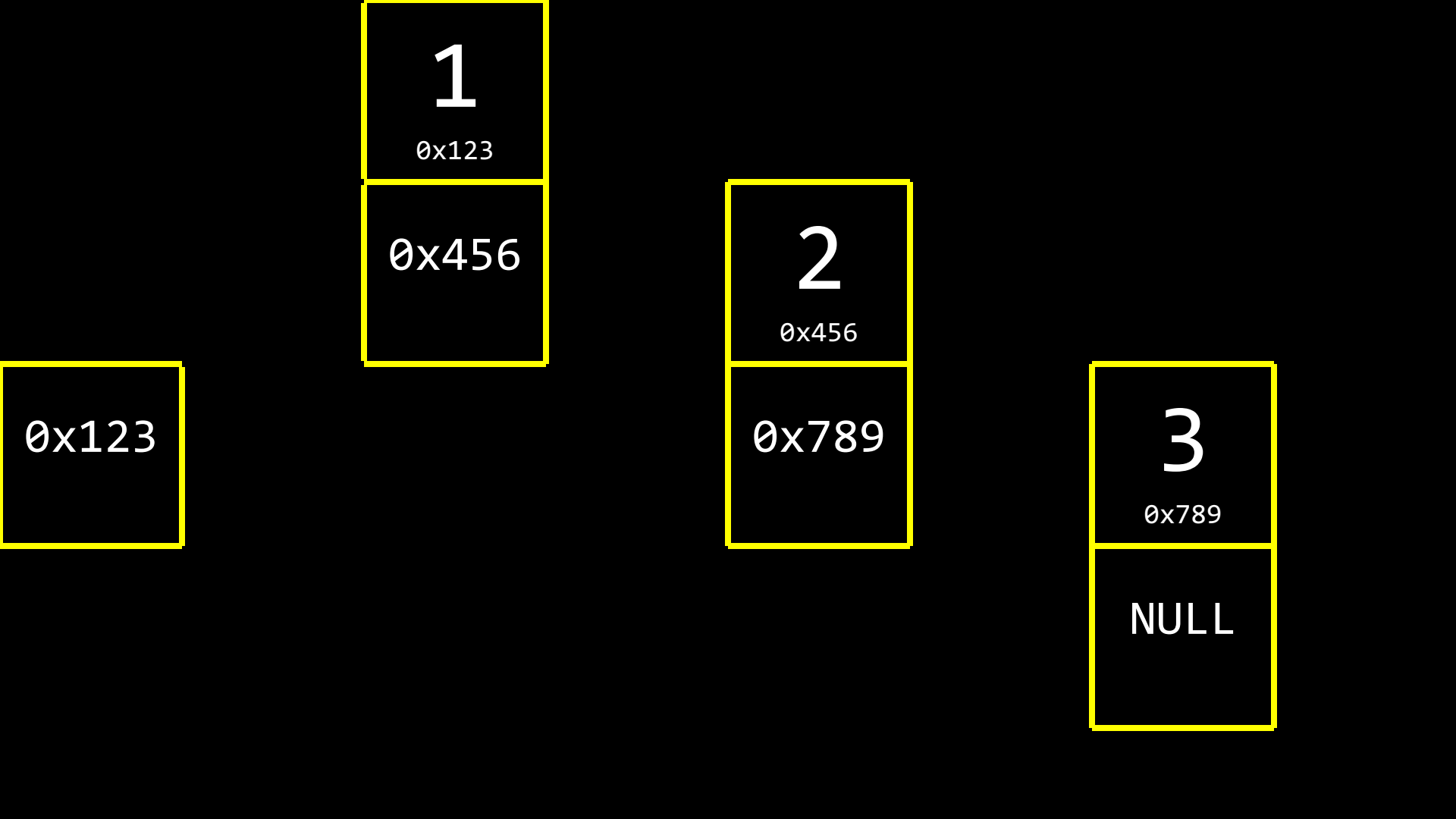

1
0x123
2
0x456
3
0x789

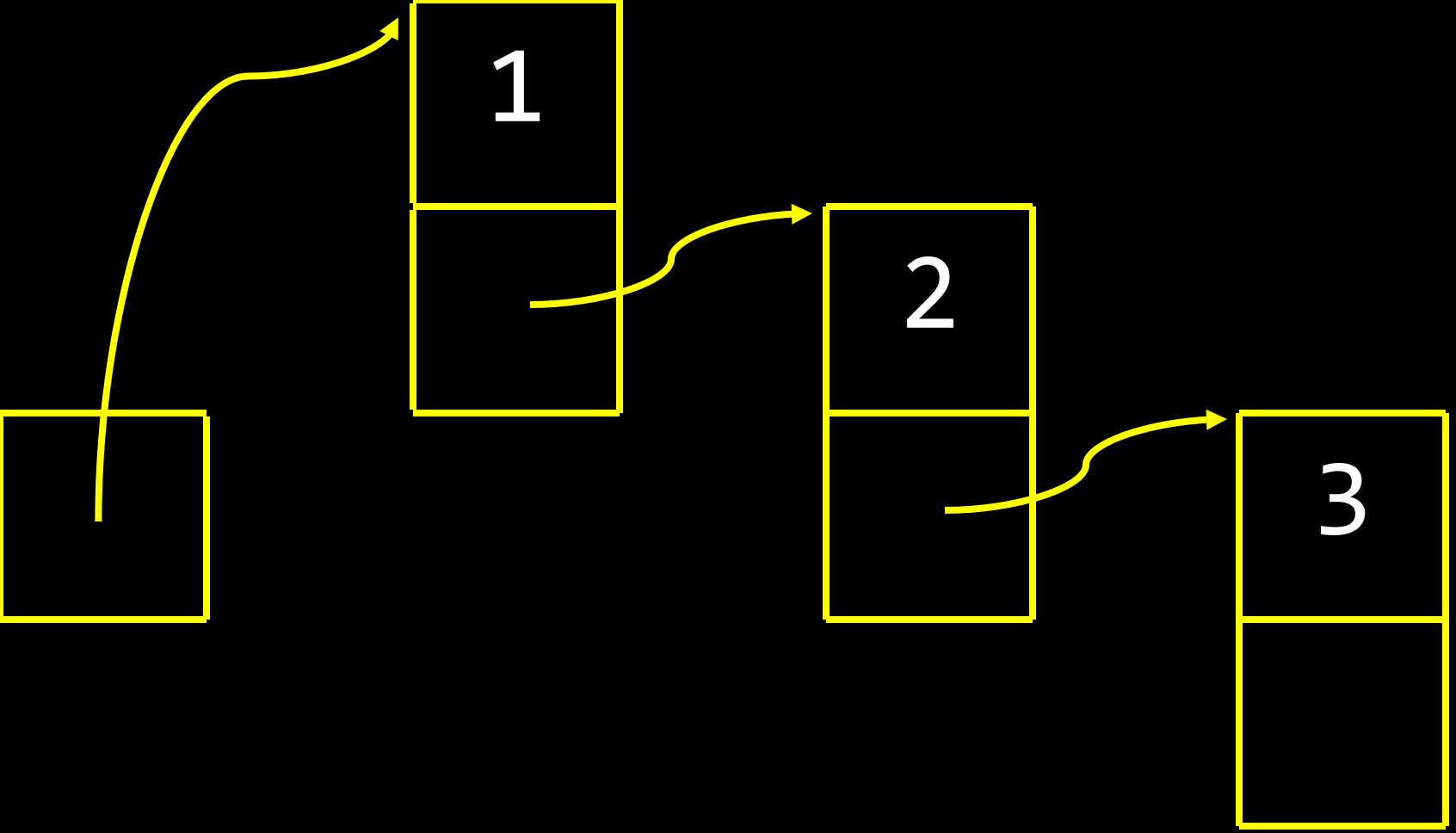

1
0x123
0x456
2
0x456
3
0x789


1
0x123
0x456
2
0x456
0x789
3
0x789


1
0x123
0x456
2
0x456
0x789
3
0x789
0x0


1
0x123
0x456
2
0x456
0x789
3
0x789
NULL


1
0x123
0x456
2
0x456
0x789
3
0x789
NULL
0x123


1
2
3

```
typedef struct
{
    string name;
    string number;
} person;
```

```
typedef struct
{
    char *name;
    char *number;
} person;
```

```
typedef struct
{


} person;
```

```
typedef struct
{


} node;
```

```
typedef struct
{
    int number;

} node;
```

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

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

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

```
node *list;
```

```
node *list;
```

list

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

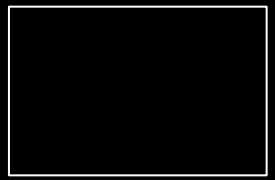
list

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

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

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

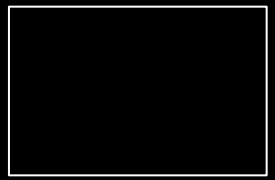
list



n

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

list

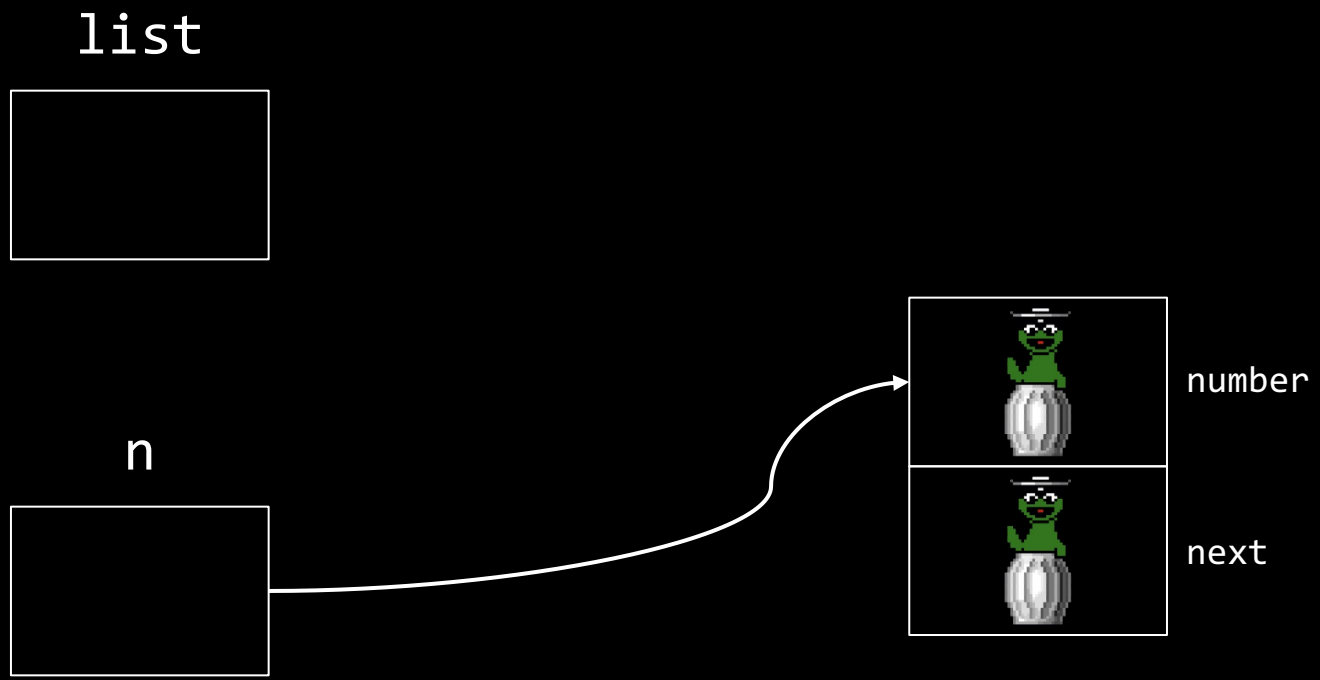


n

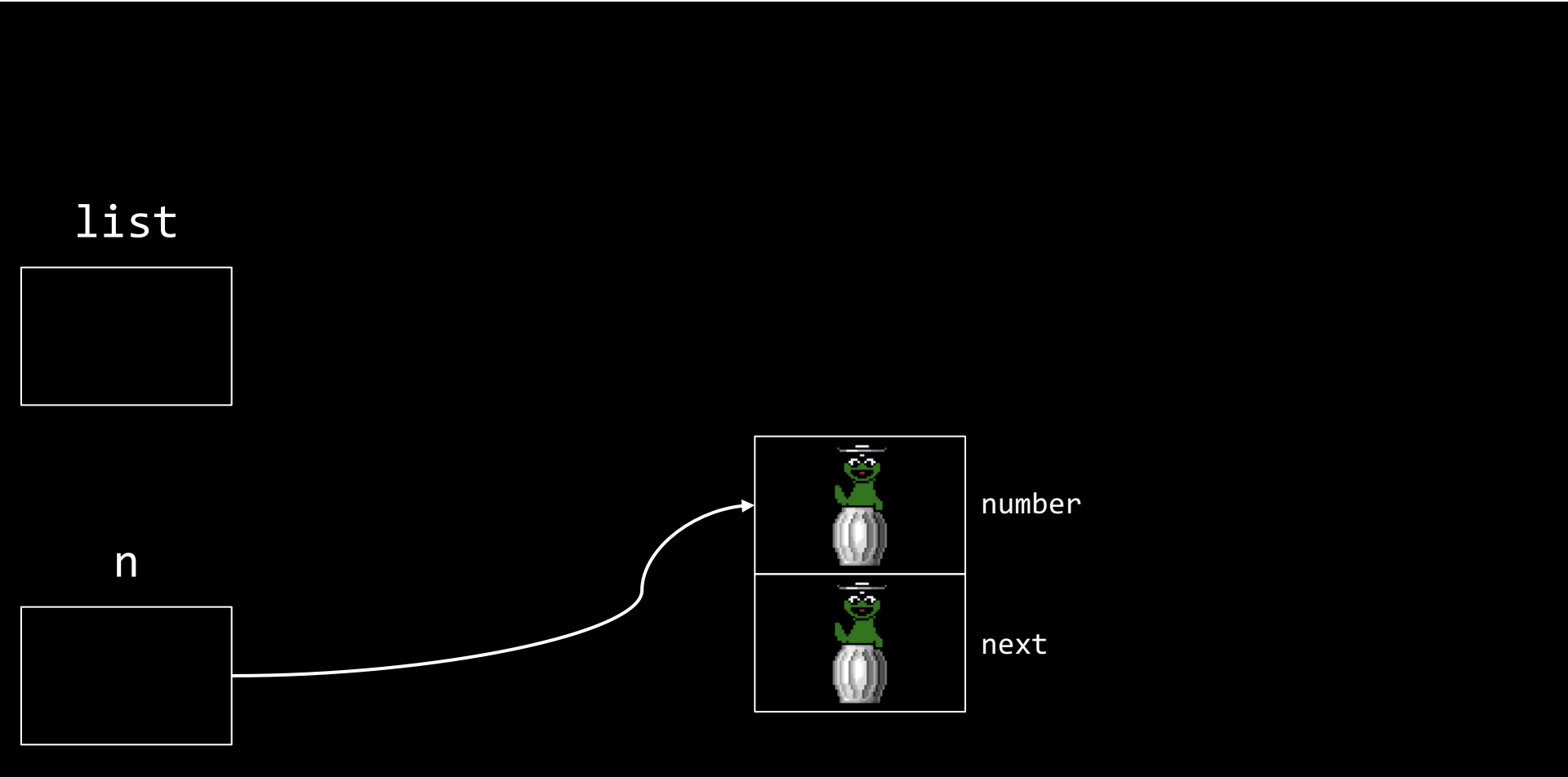
number

next

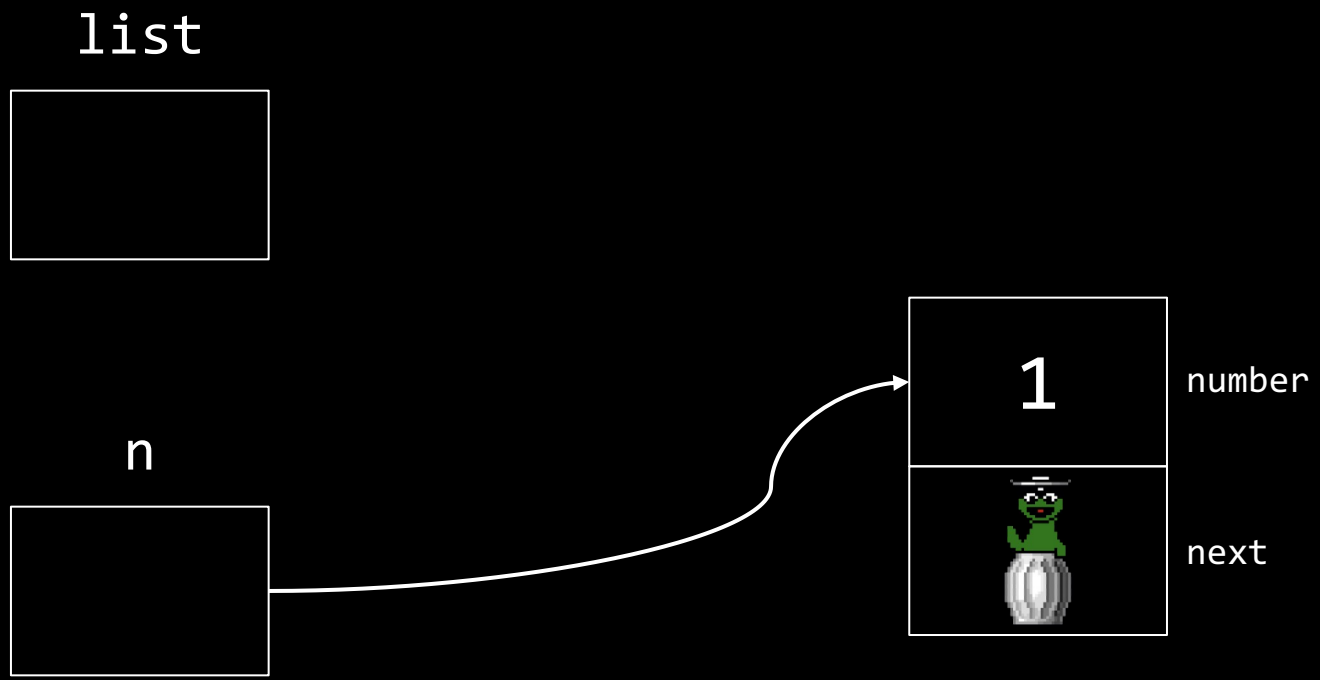
```
node *n = malloc(sizeof(node));
```



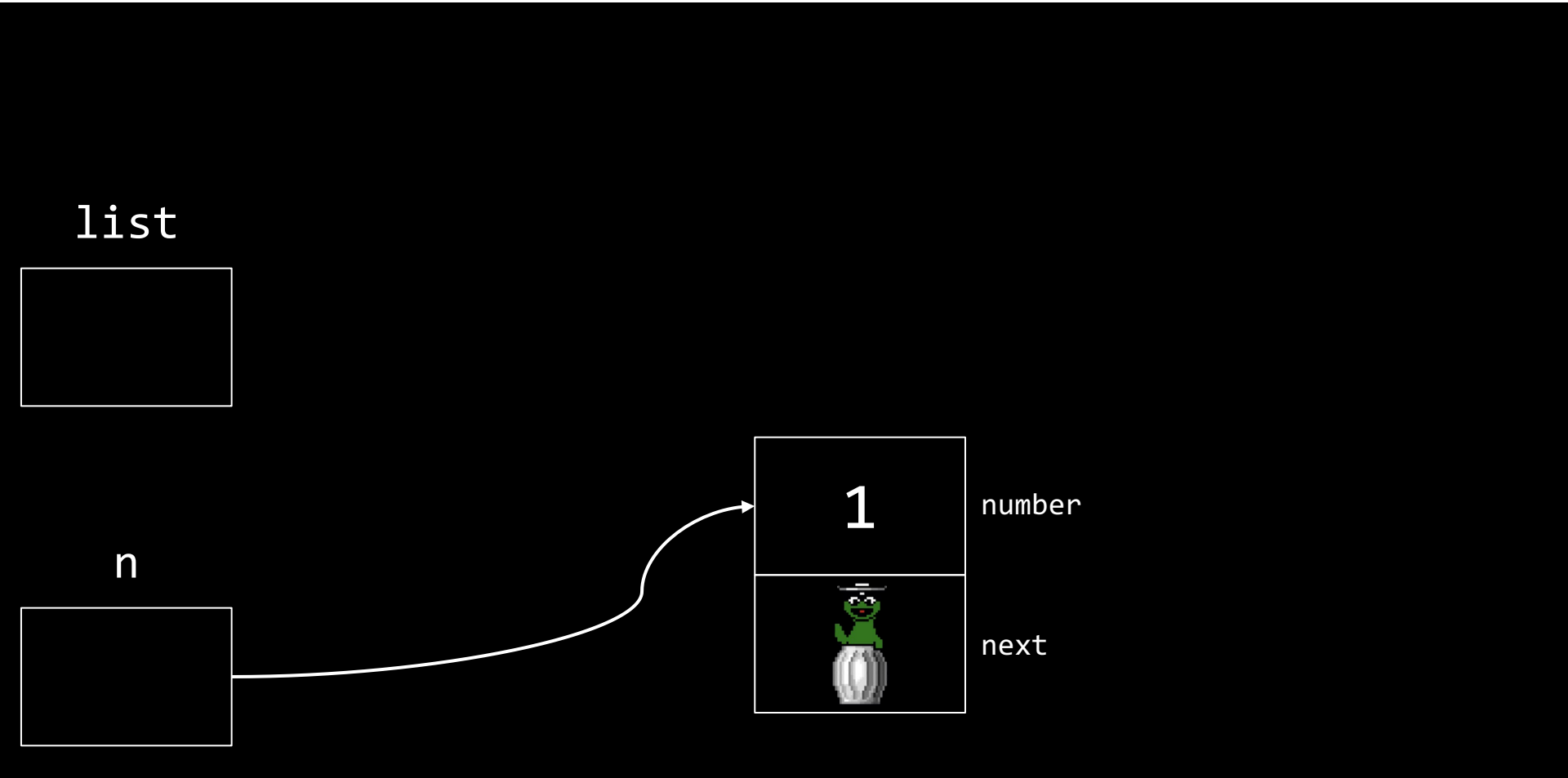
```
(*n).number = 1;
```



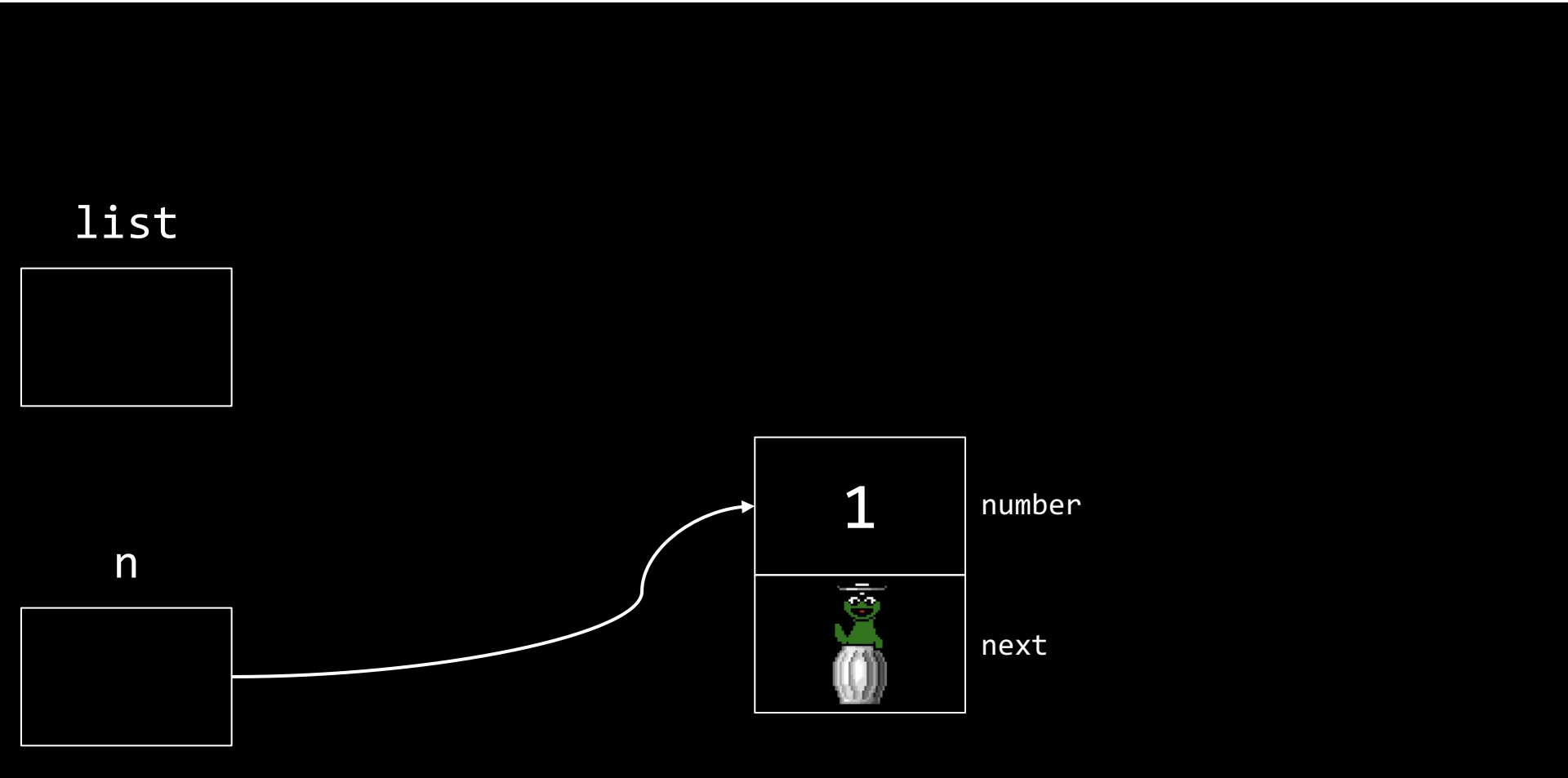
```
(*n).number = 1;
```



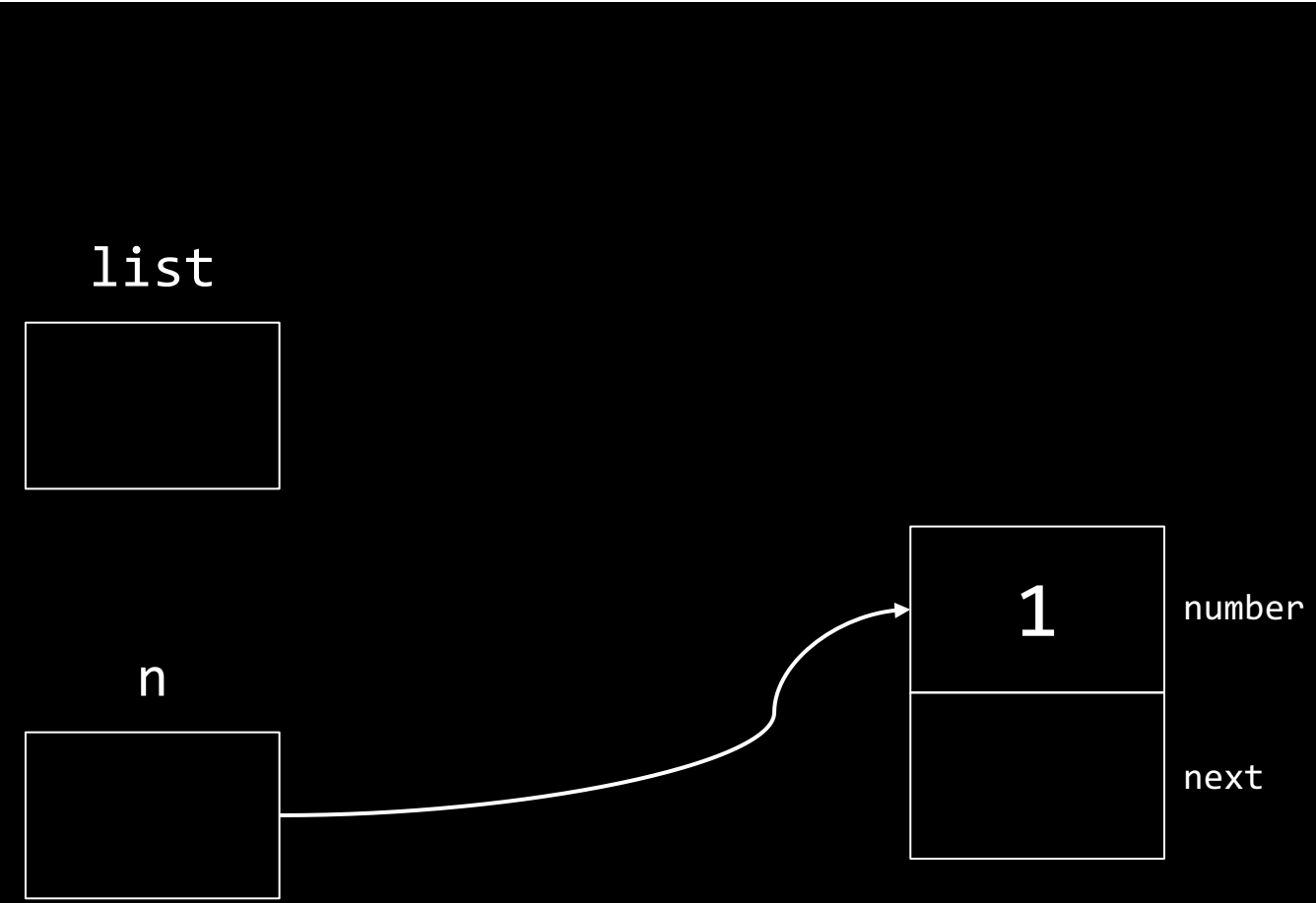
```
n->number = 1;
```



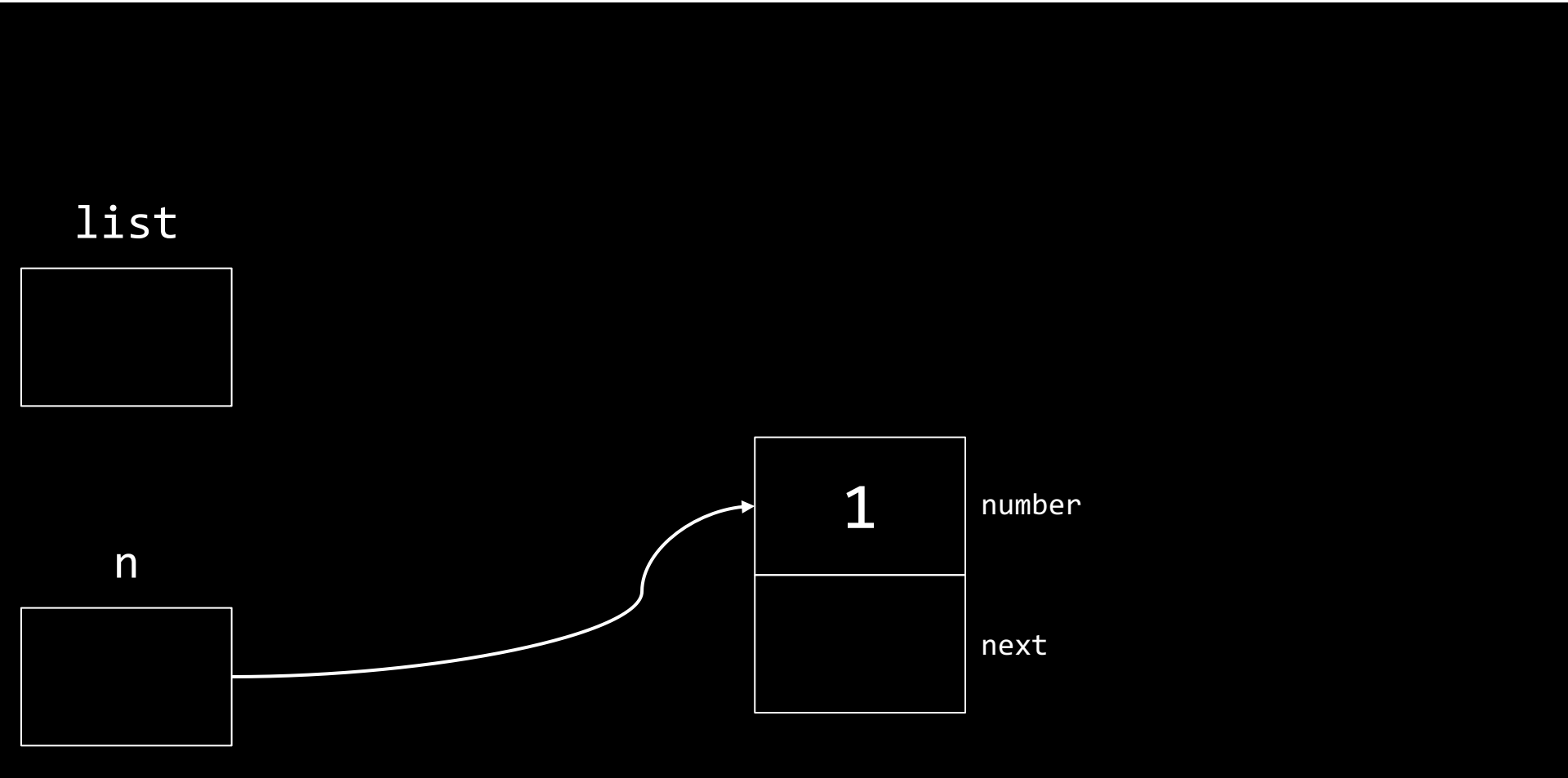
```
n->next = NULL;
```



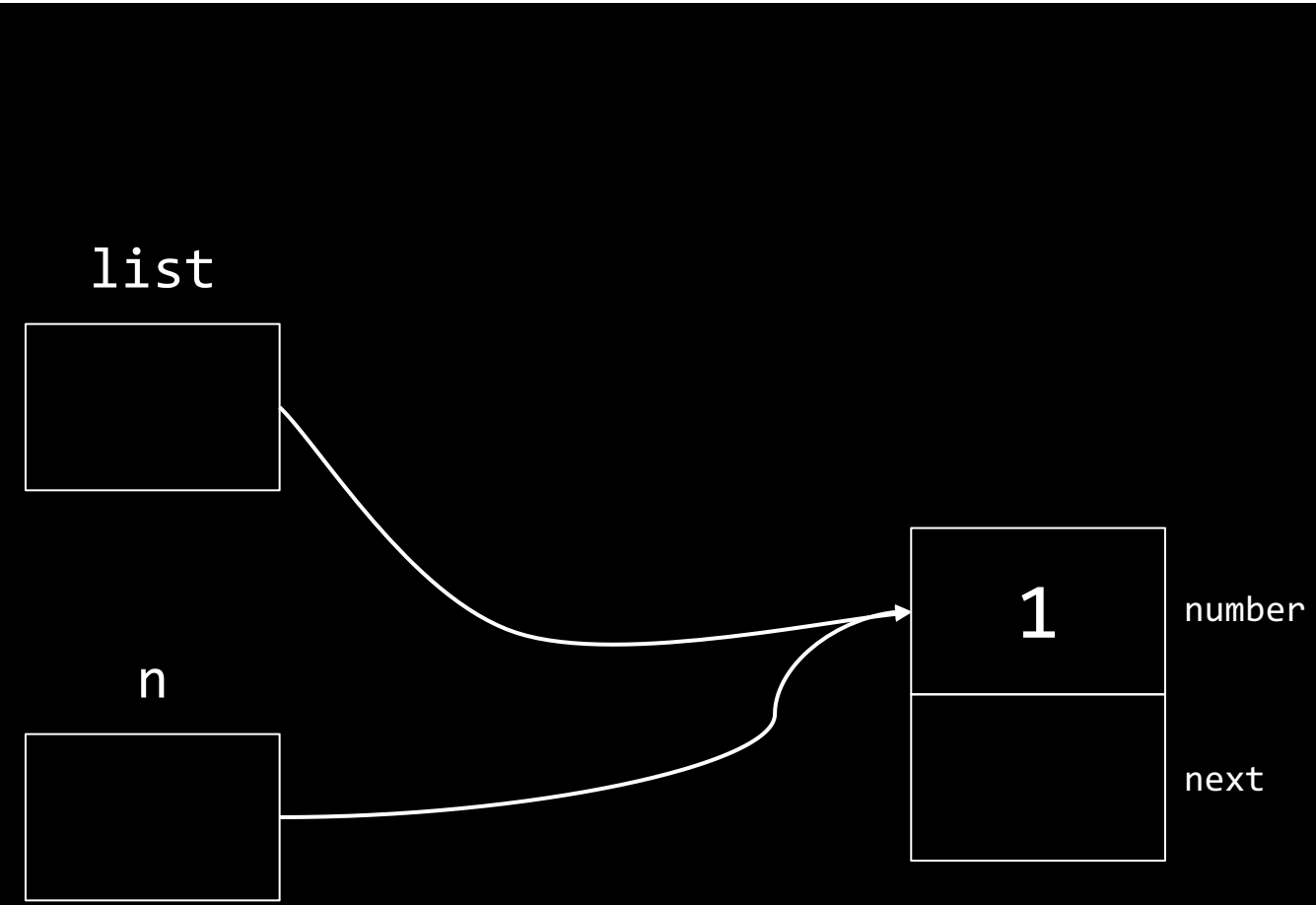
```
n->next = NULL;
```

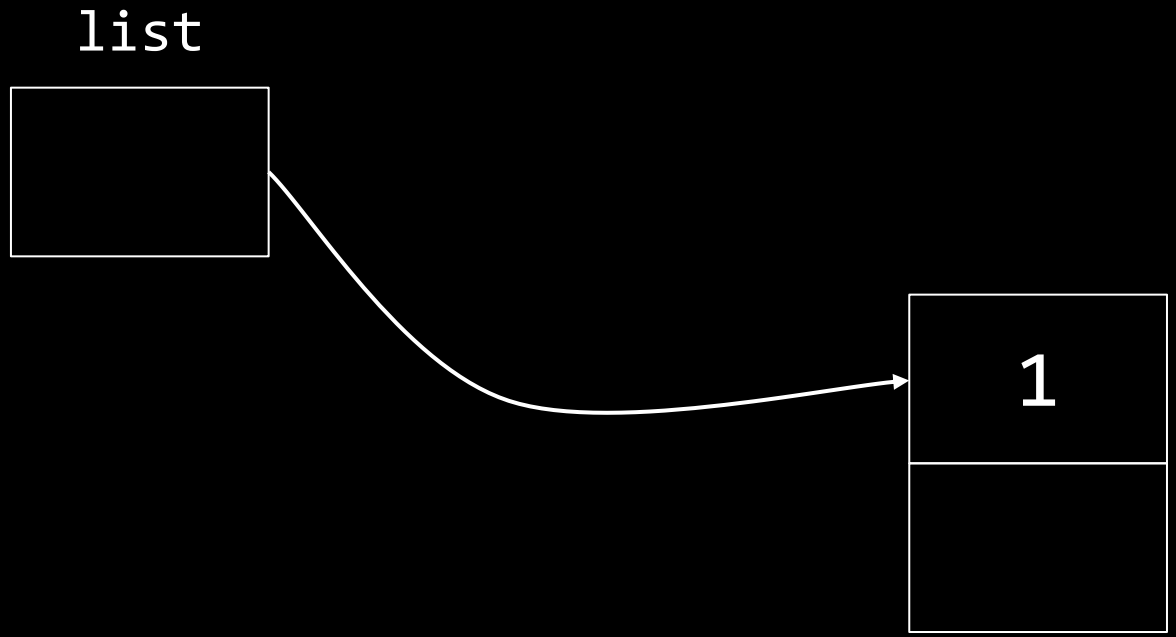


```
list = n;
```

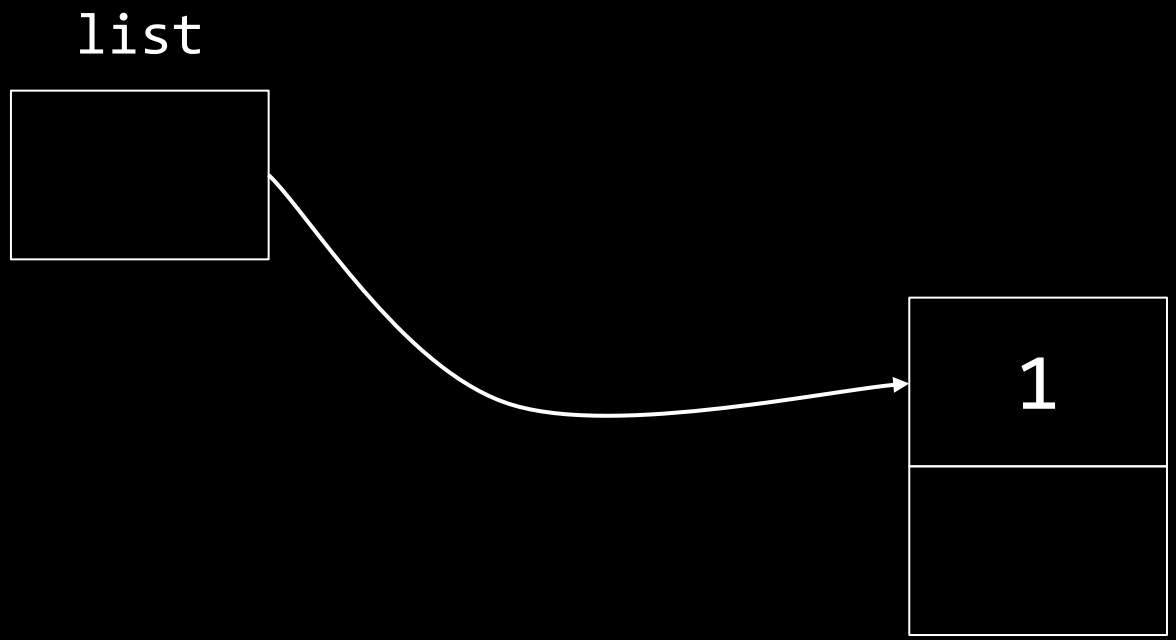


```
list = n;
```

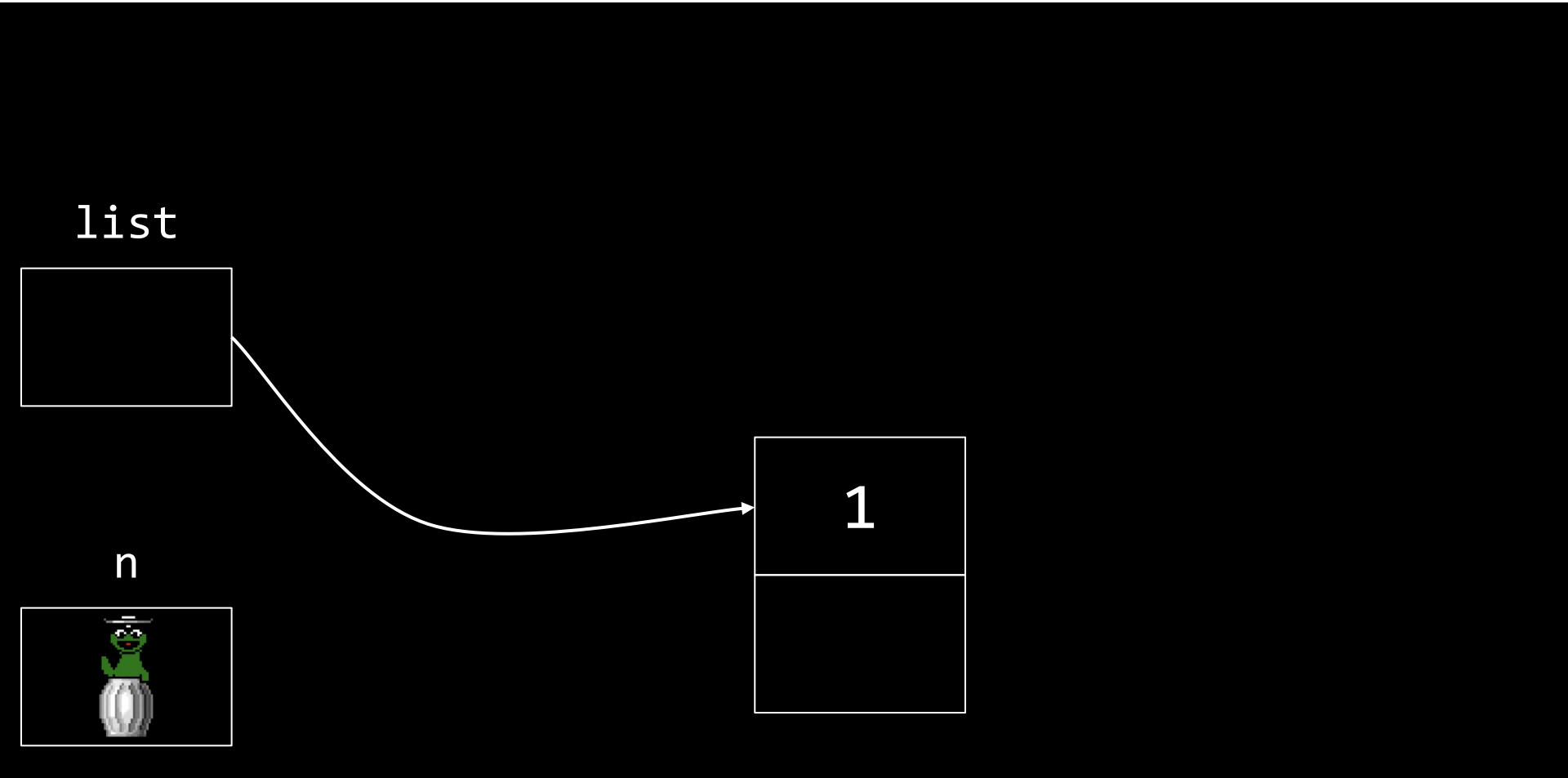



list
1

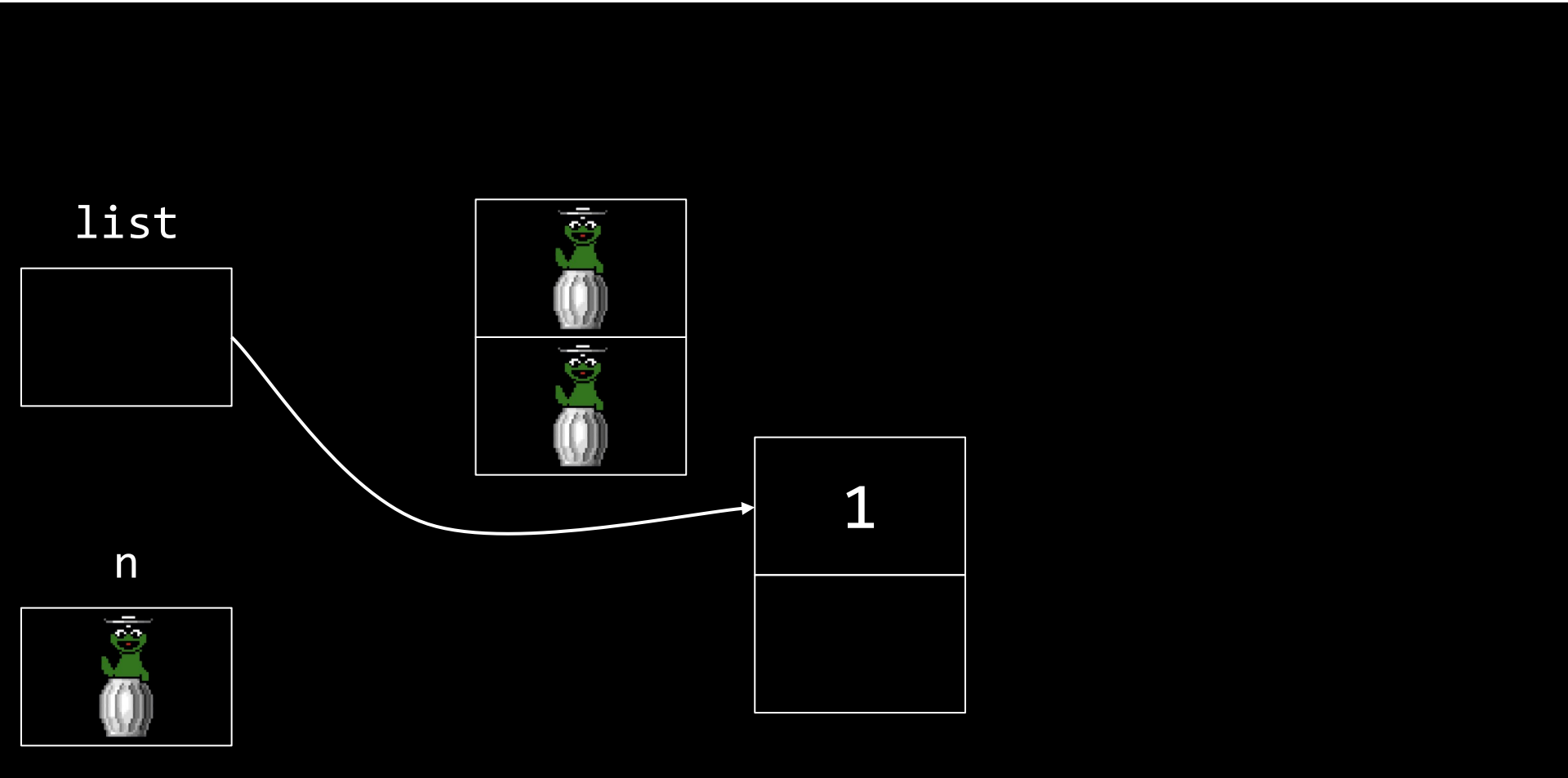
```
node *n = malloc(sizeof(node));
```



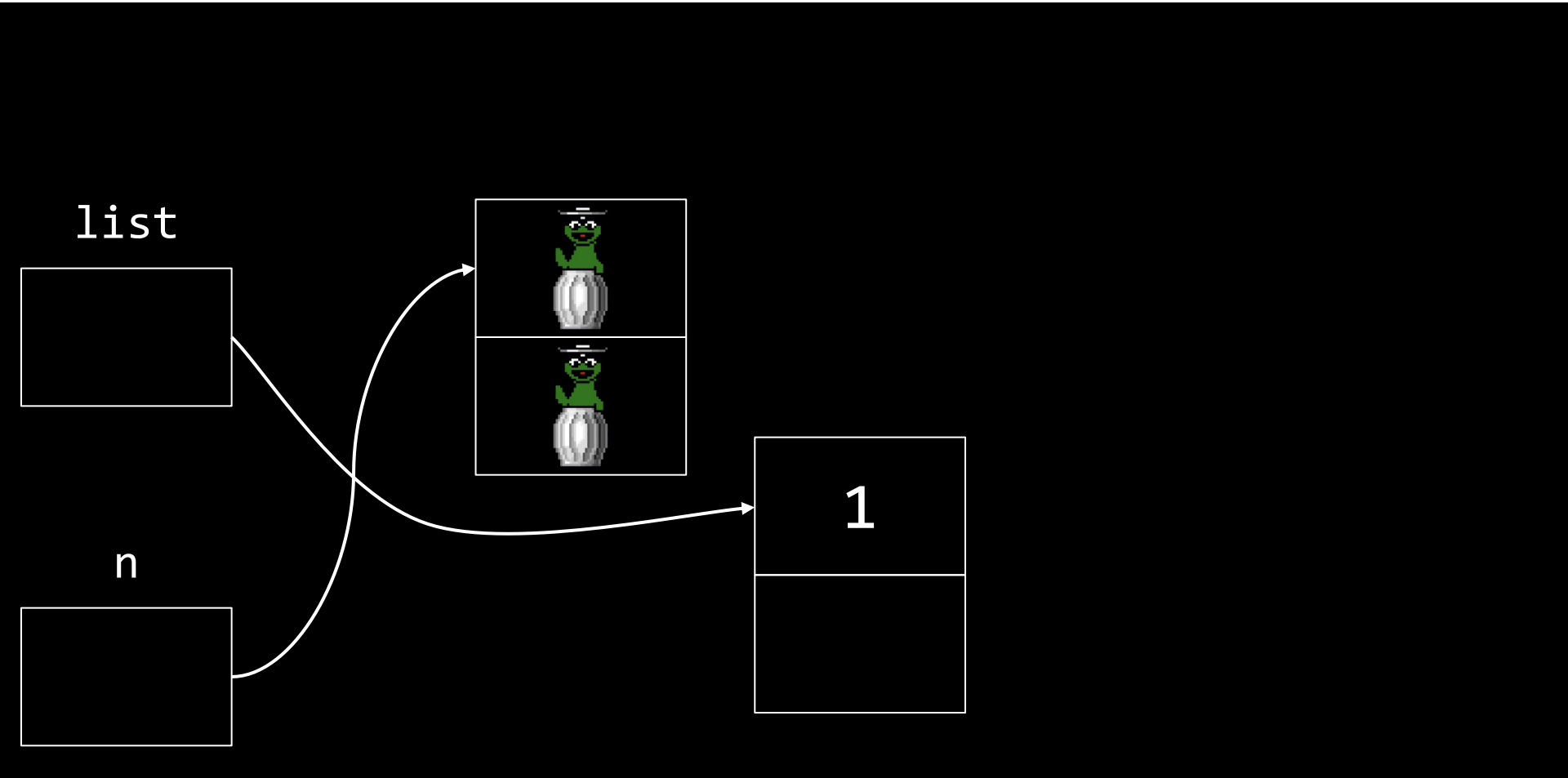
```
node *n = malloc(sizeof(node));
```



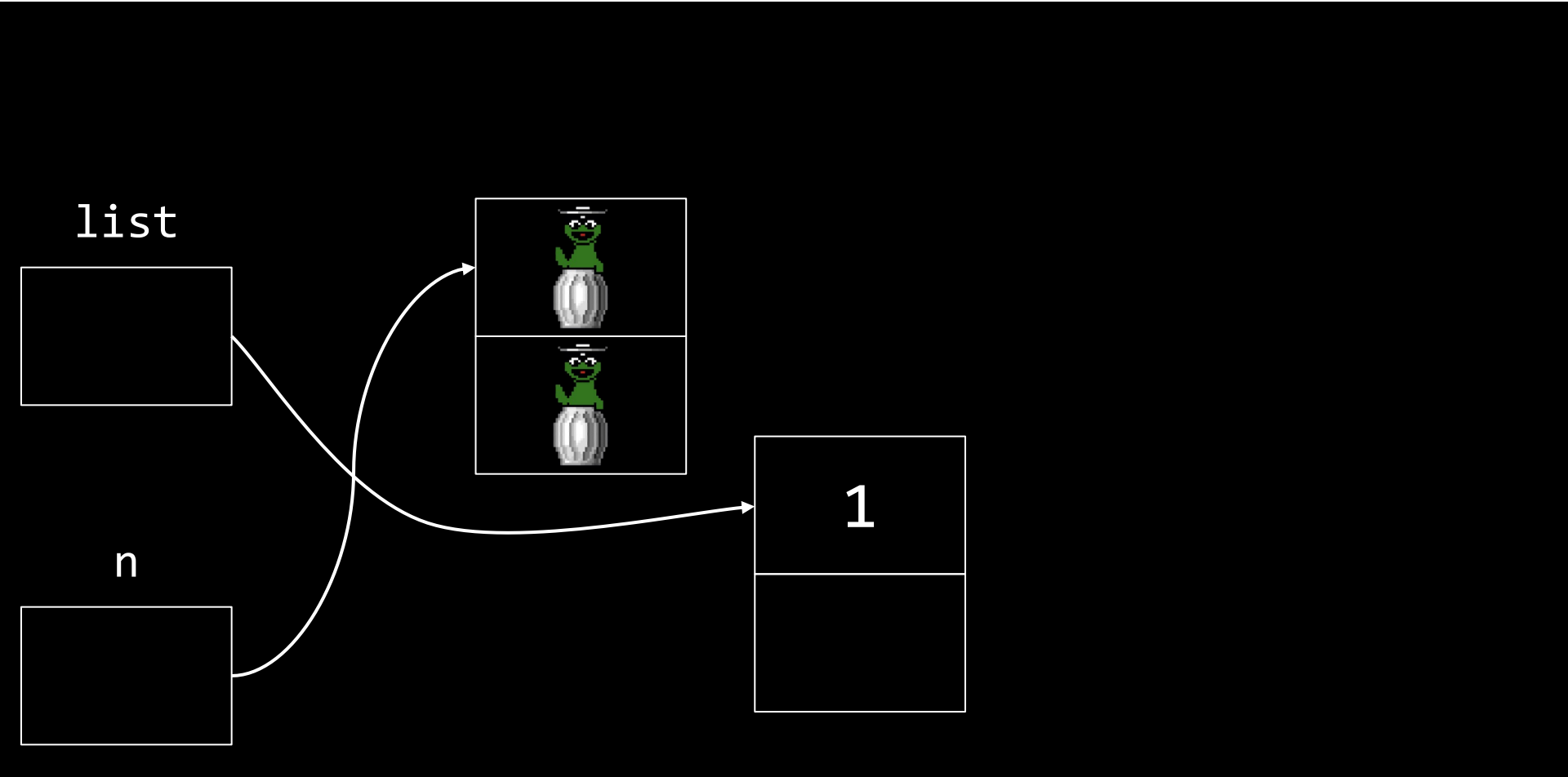
```
node *n = malloc(sizeof(node));
```



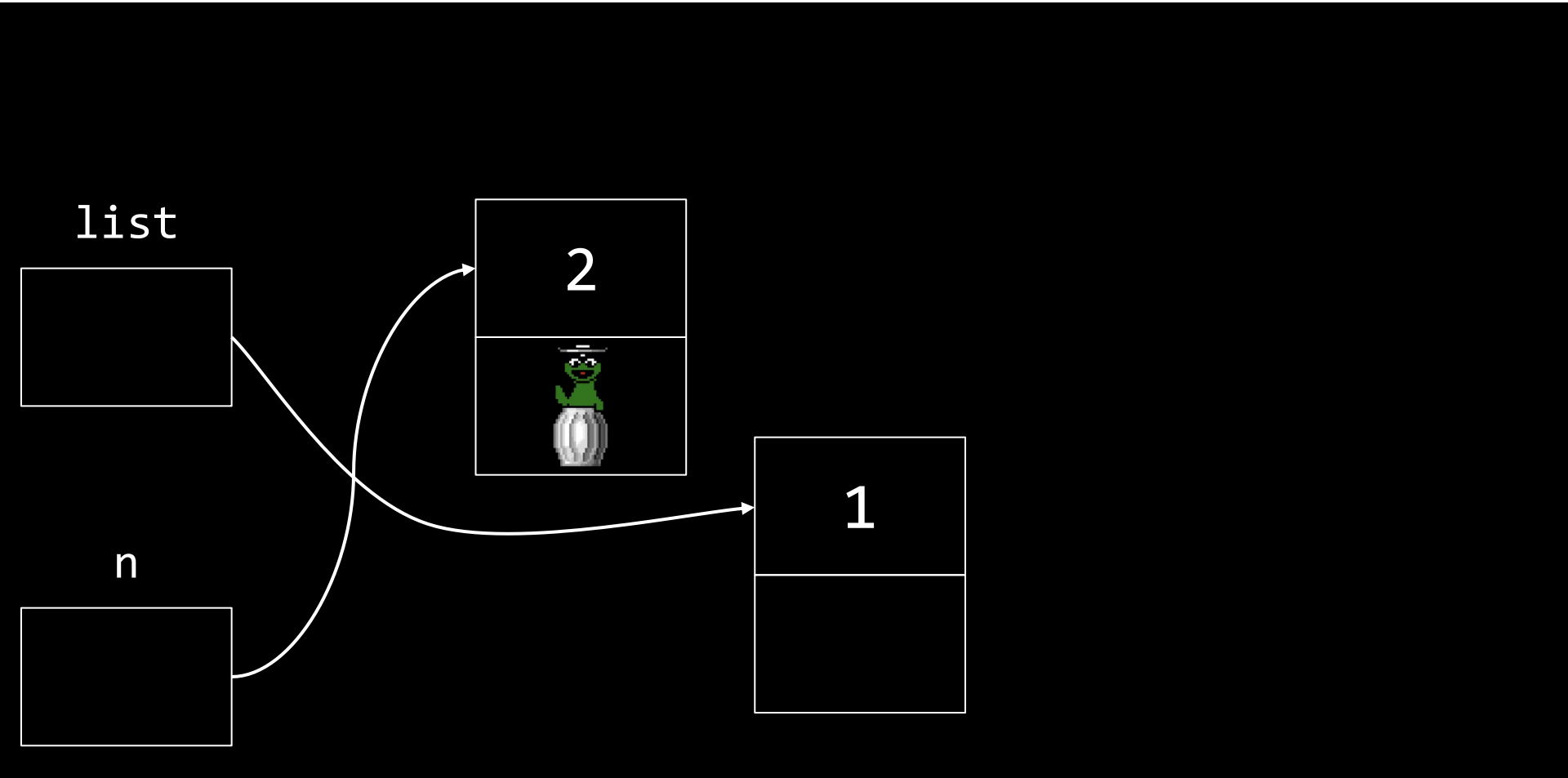
```
node *n = malloc(sizeof(node));
```



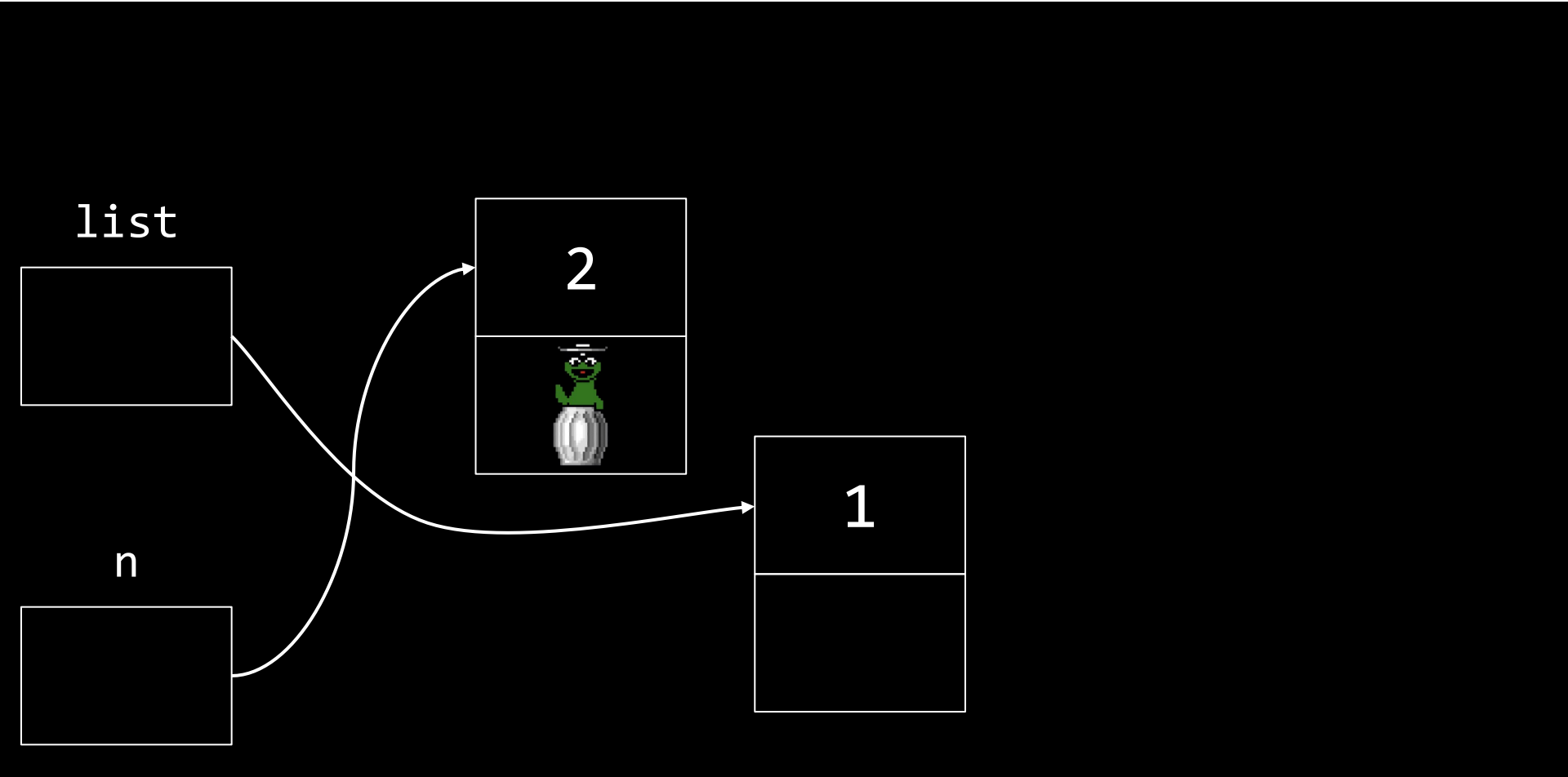
```
n->number = 2;
```



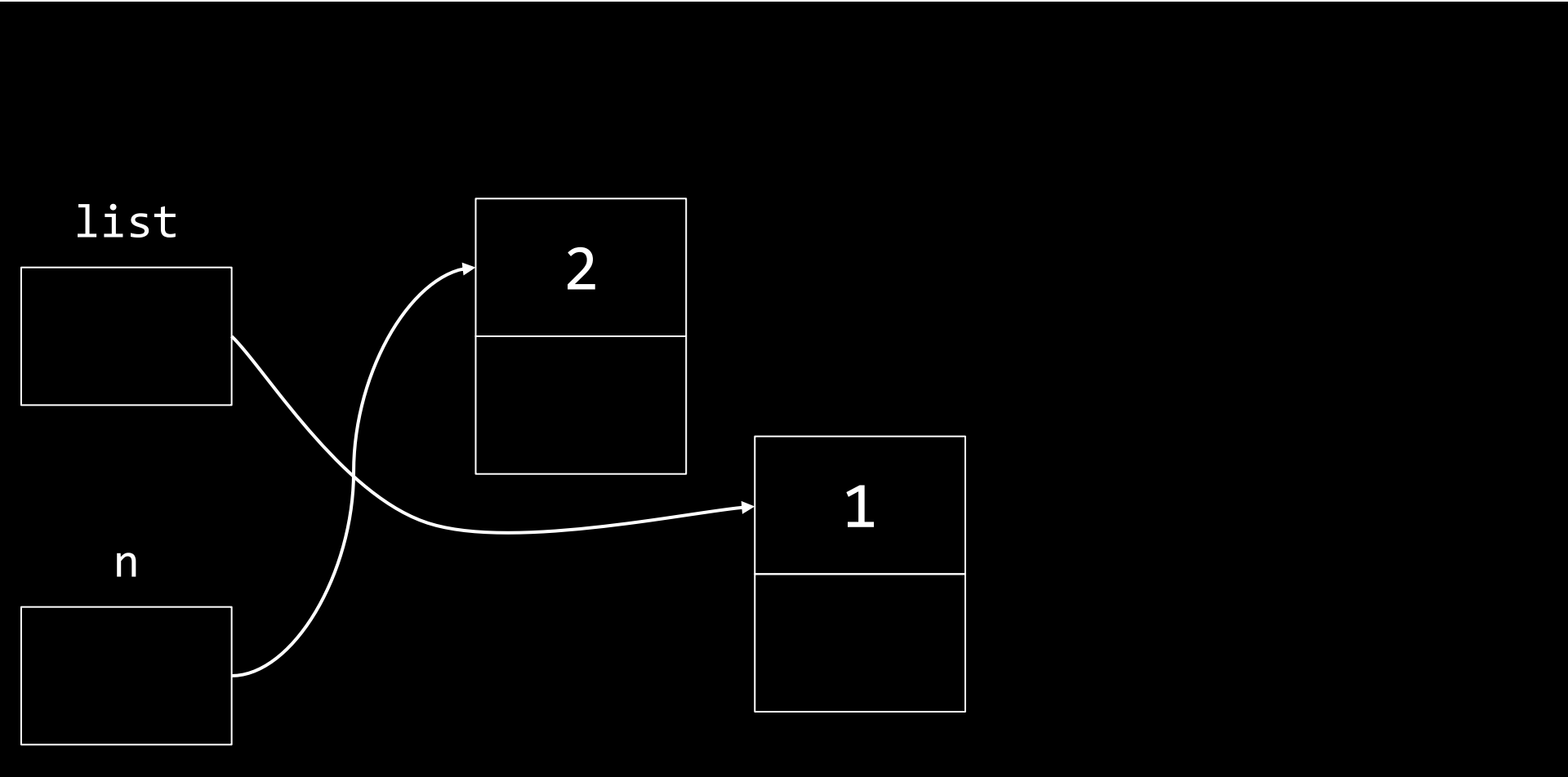
```
n->number = 2;
```

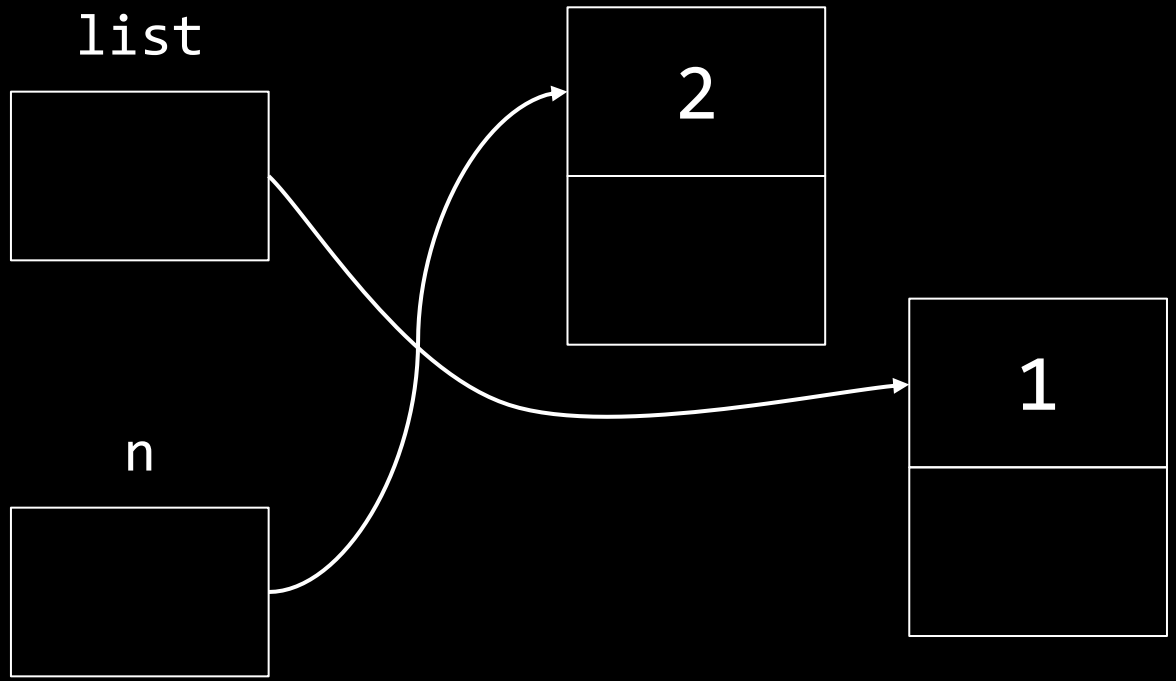


```
n->next = NULL;
```

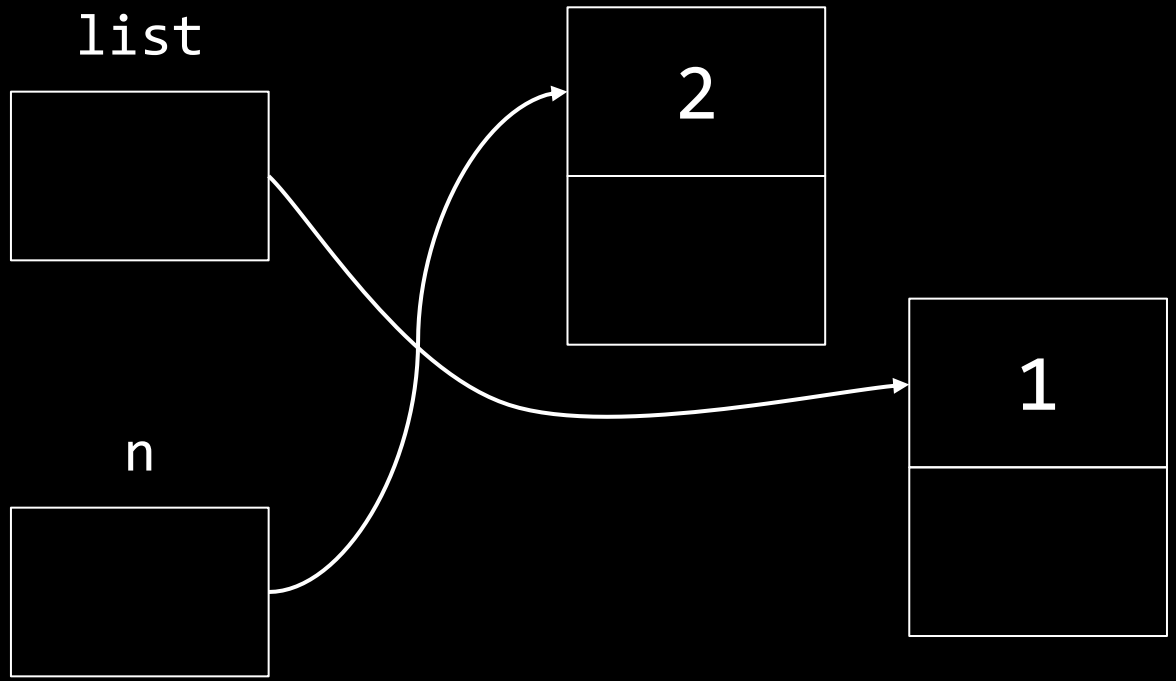


```
n->next = NULL;
```

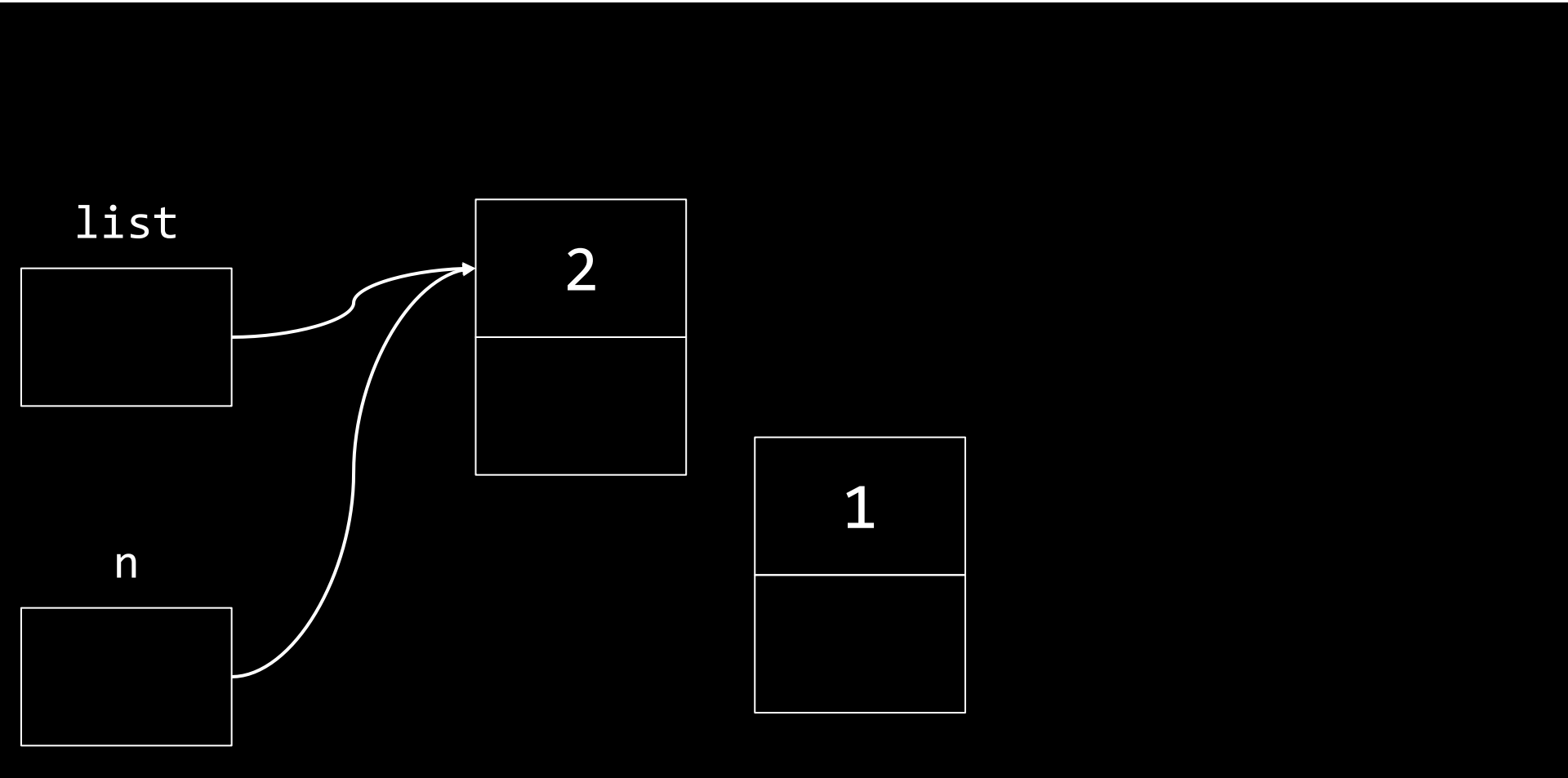



list
2
1
n

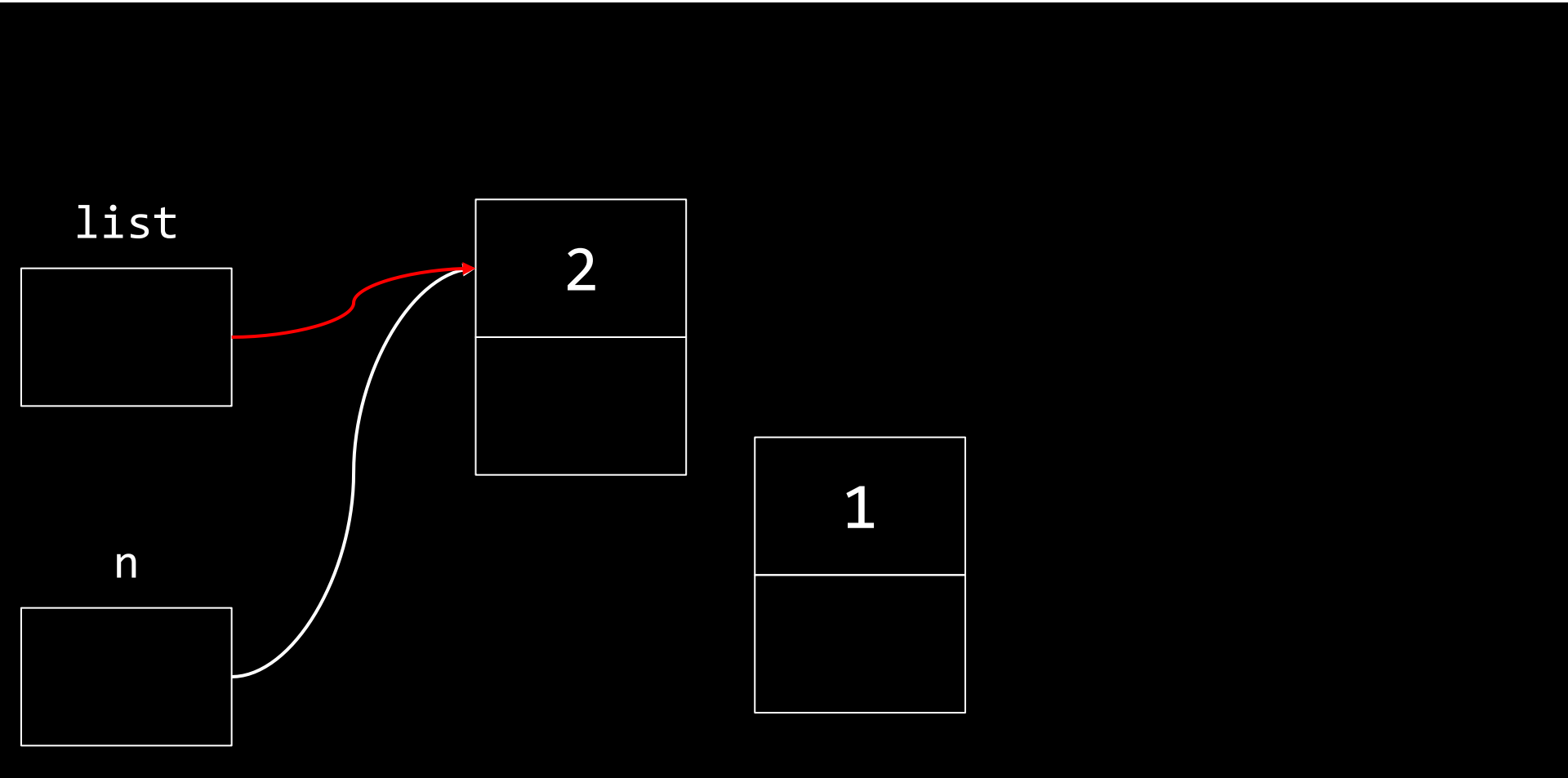
```
list = n;
```

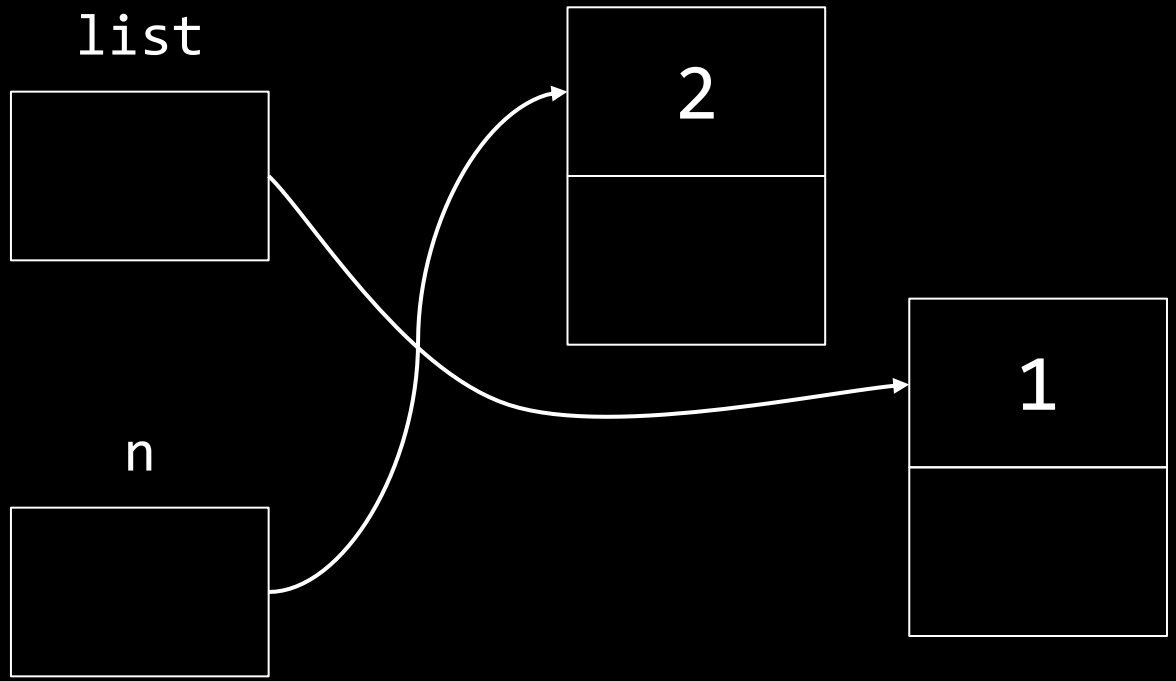


```
list = n;
```

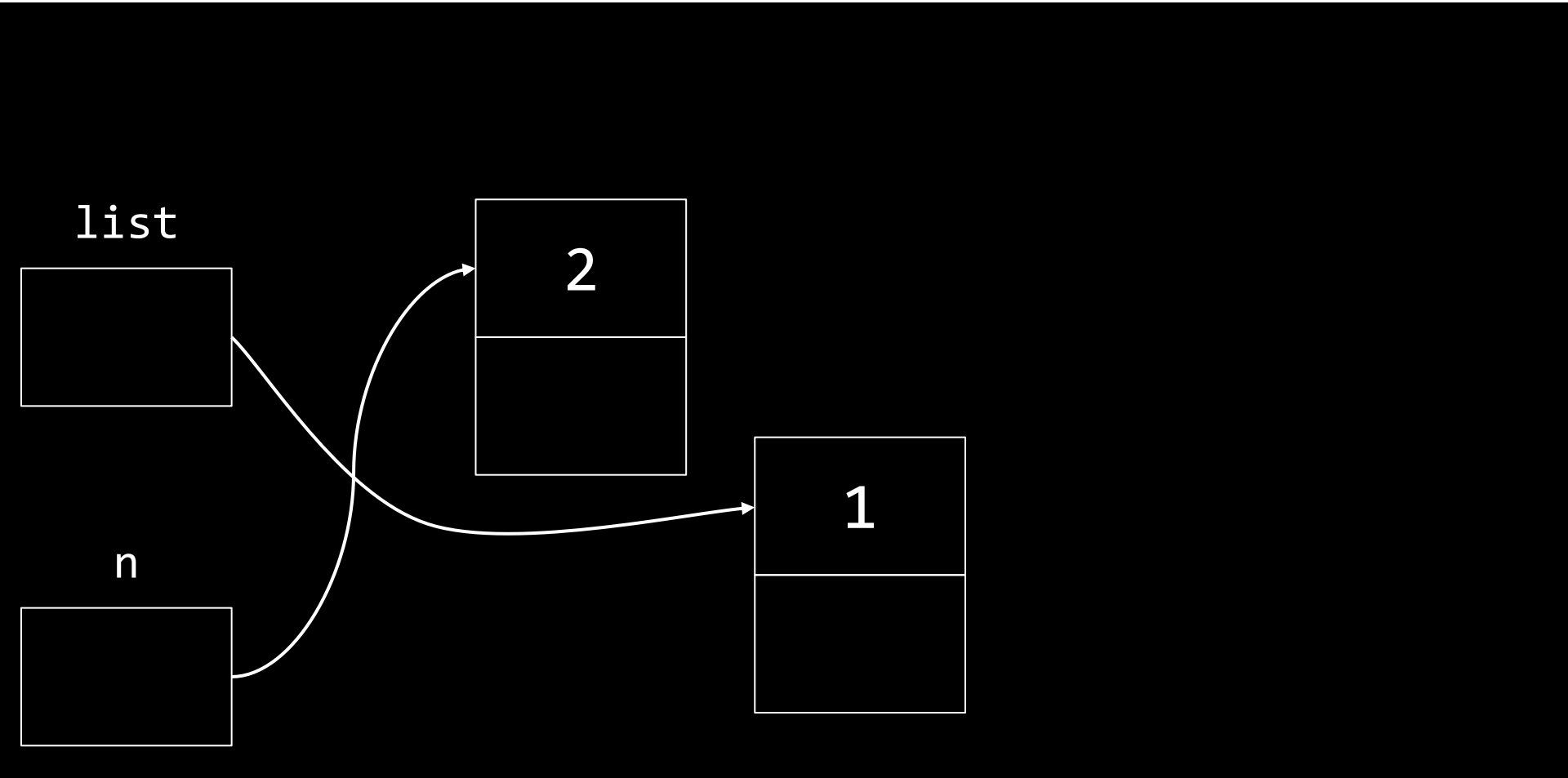


```
list = n;
```

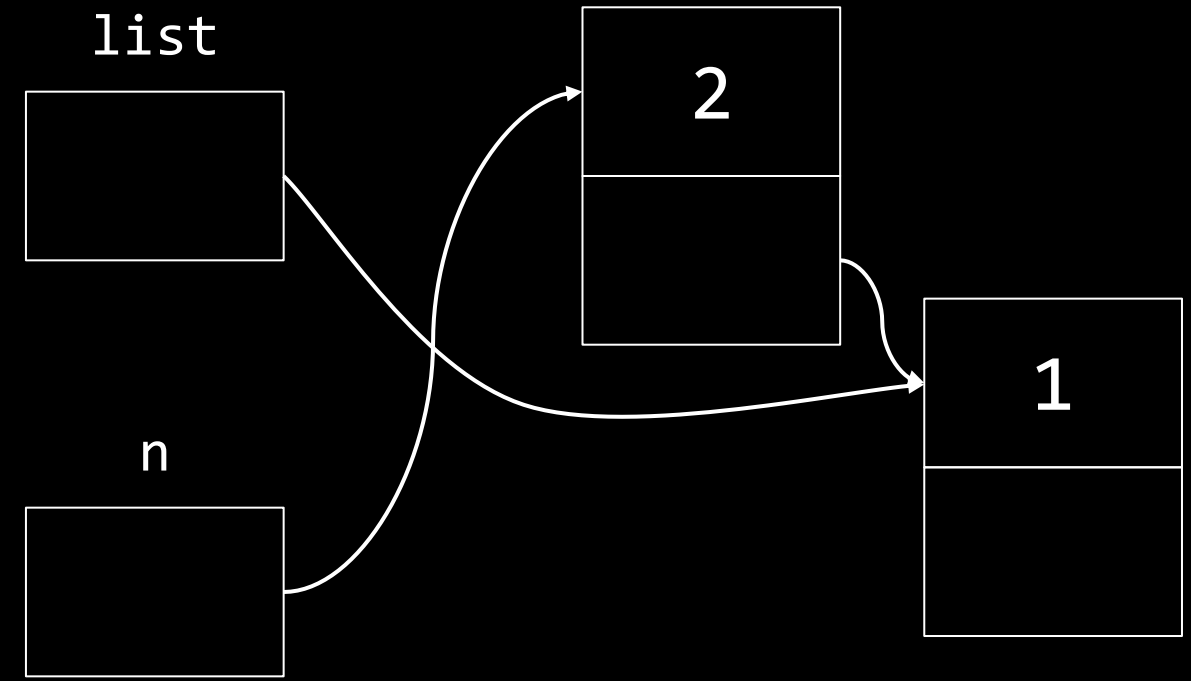



list
2
1
n

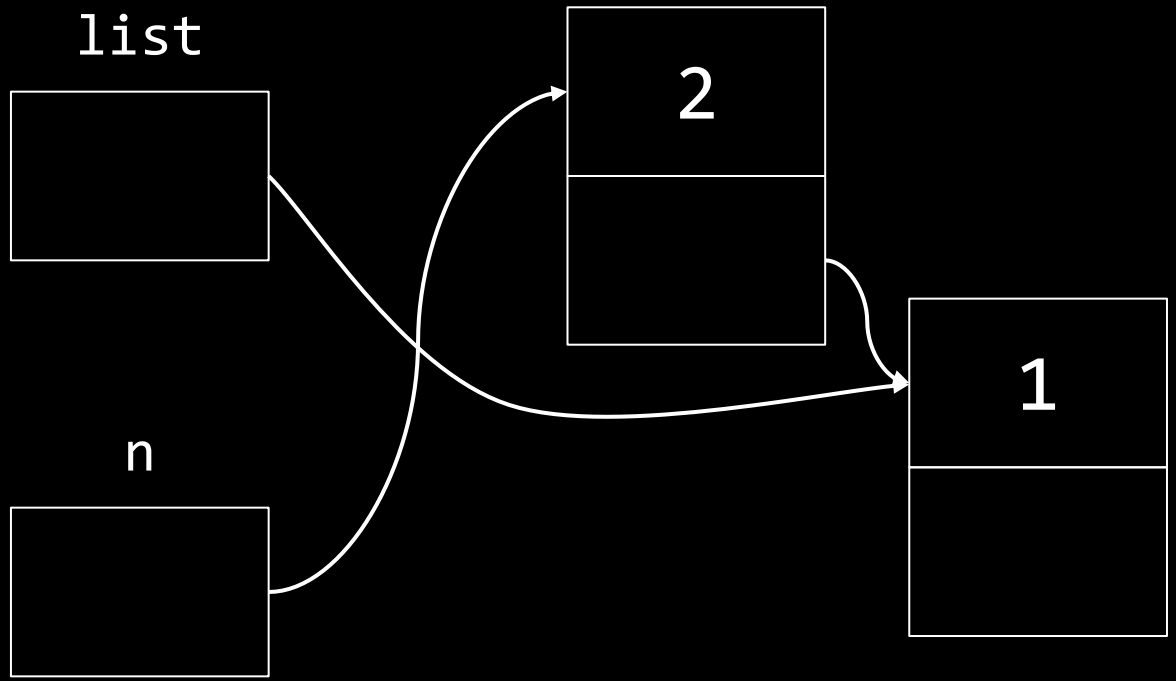
```
n->next = list;
```



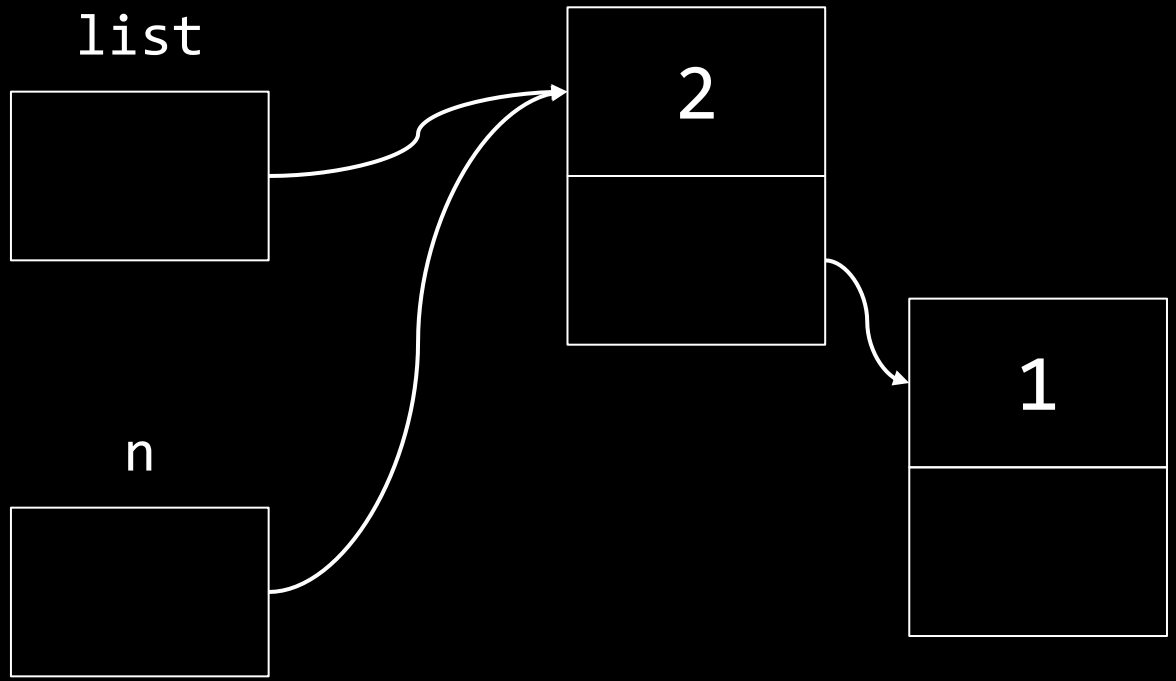
```
n->next = list;
```

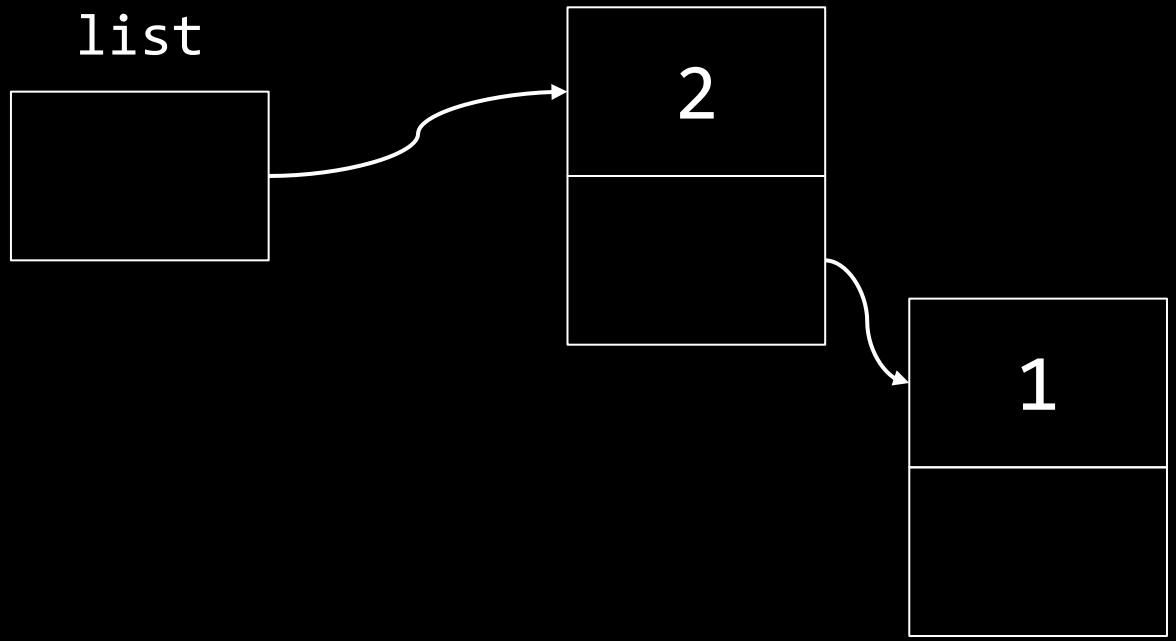


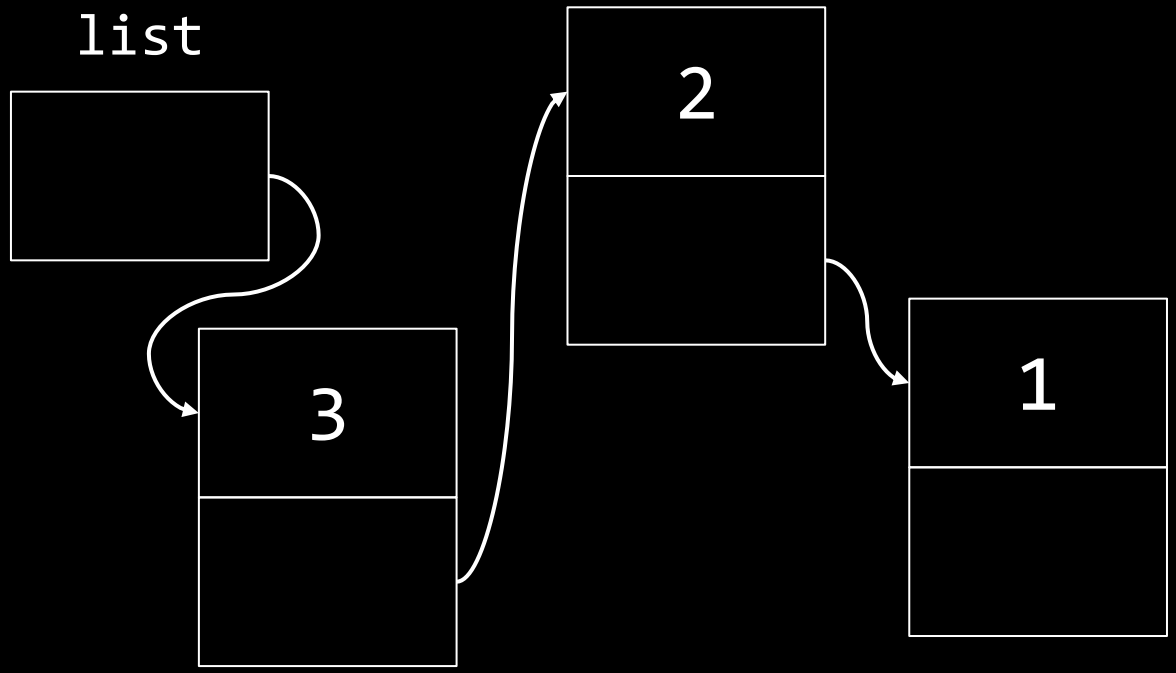
```
list = n;
```

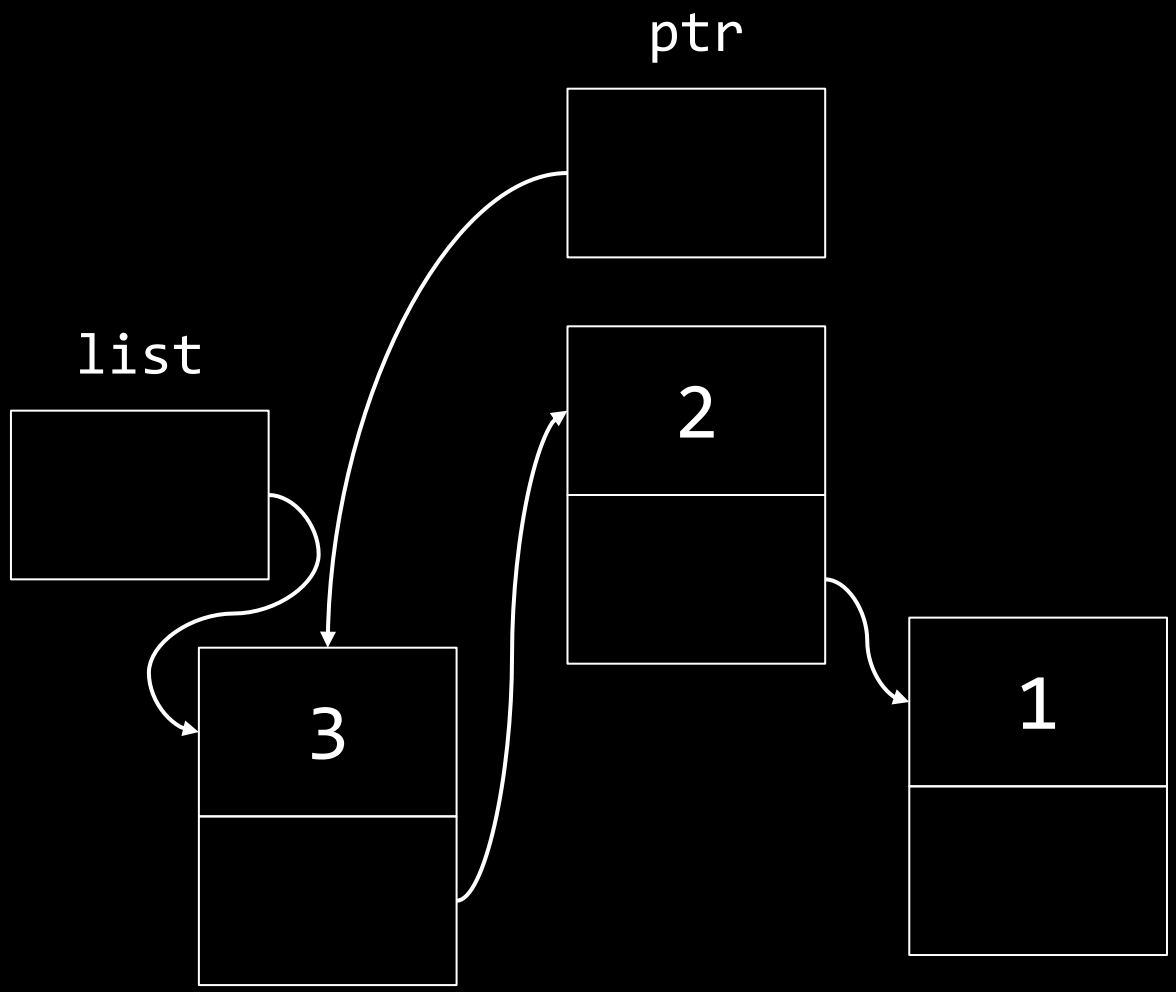


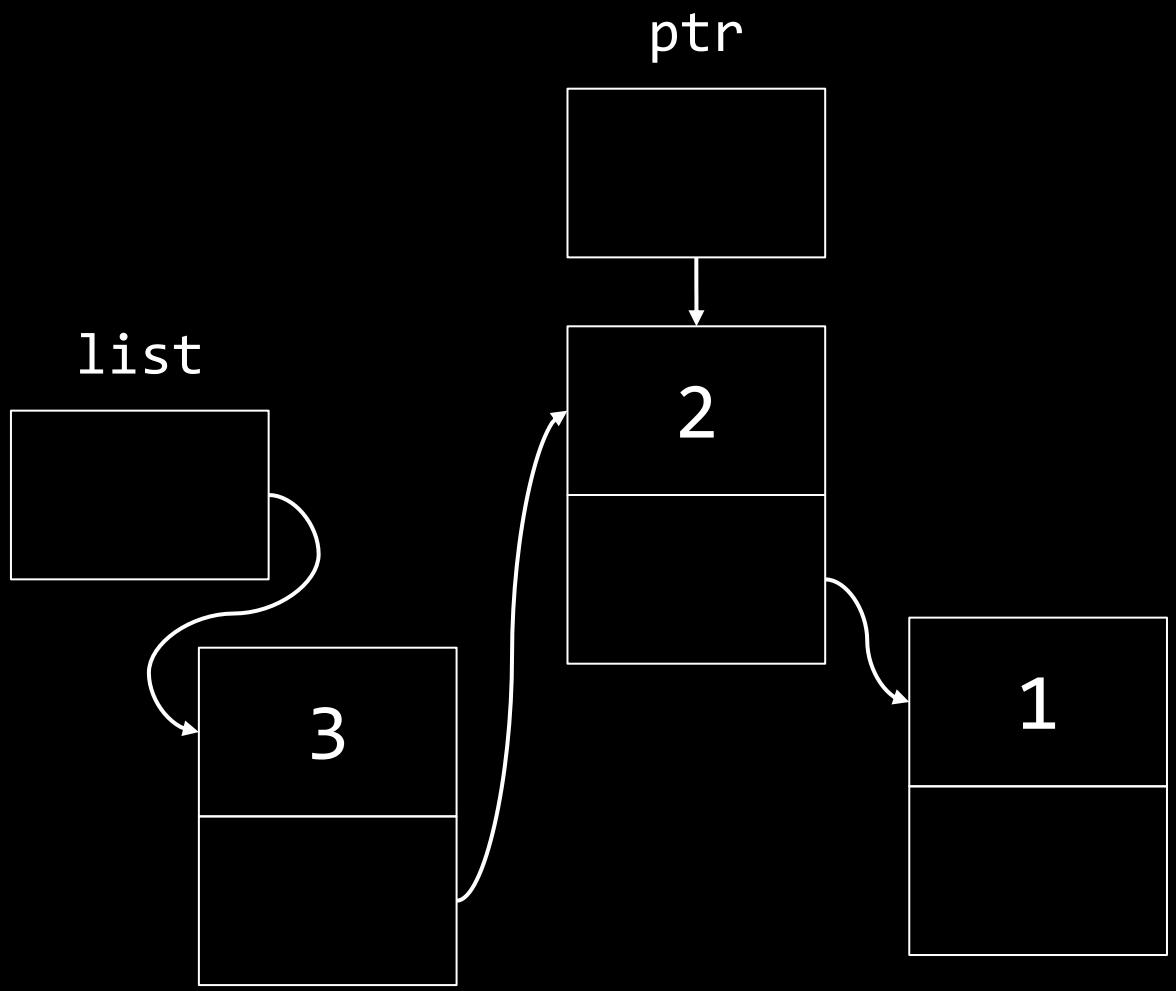
```
list = n;
```

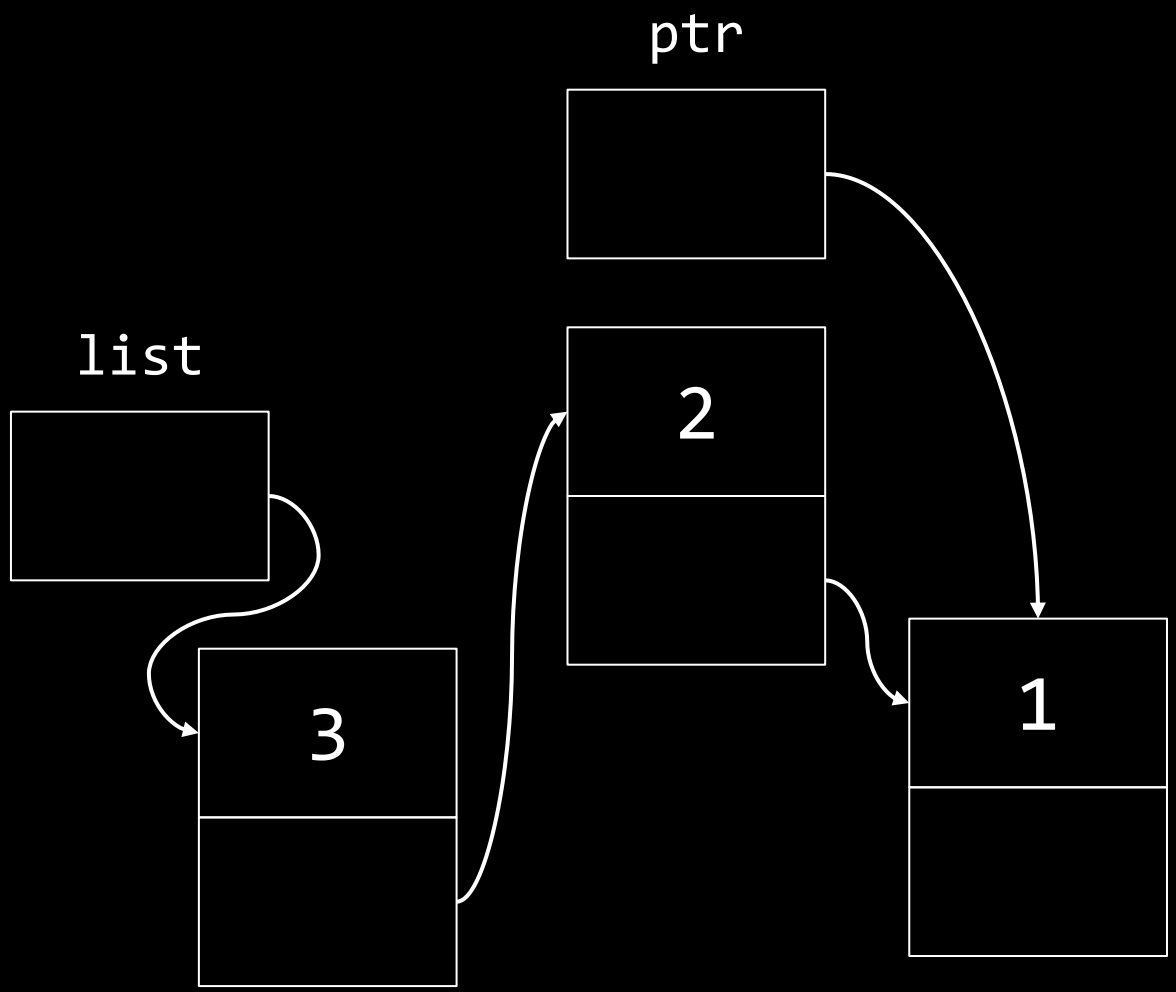


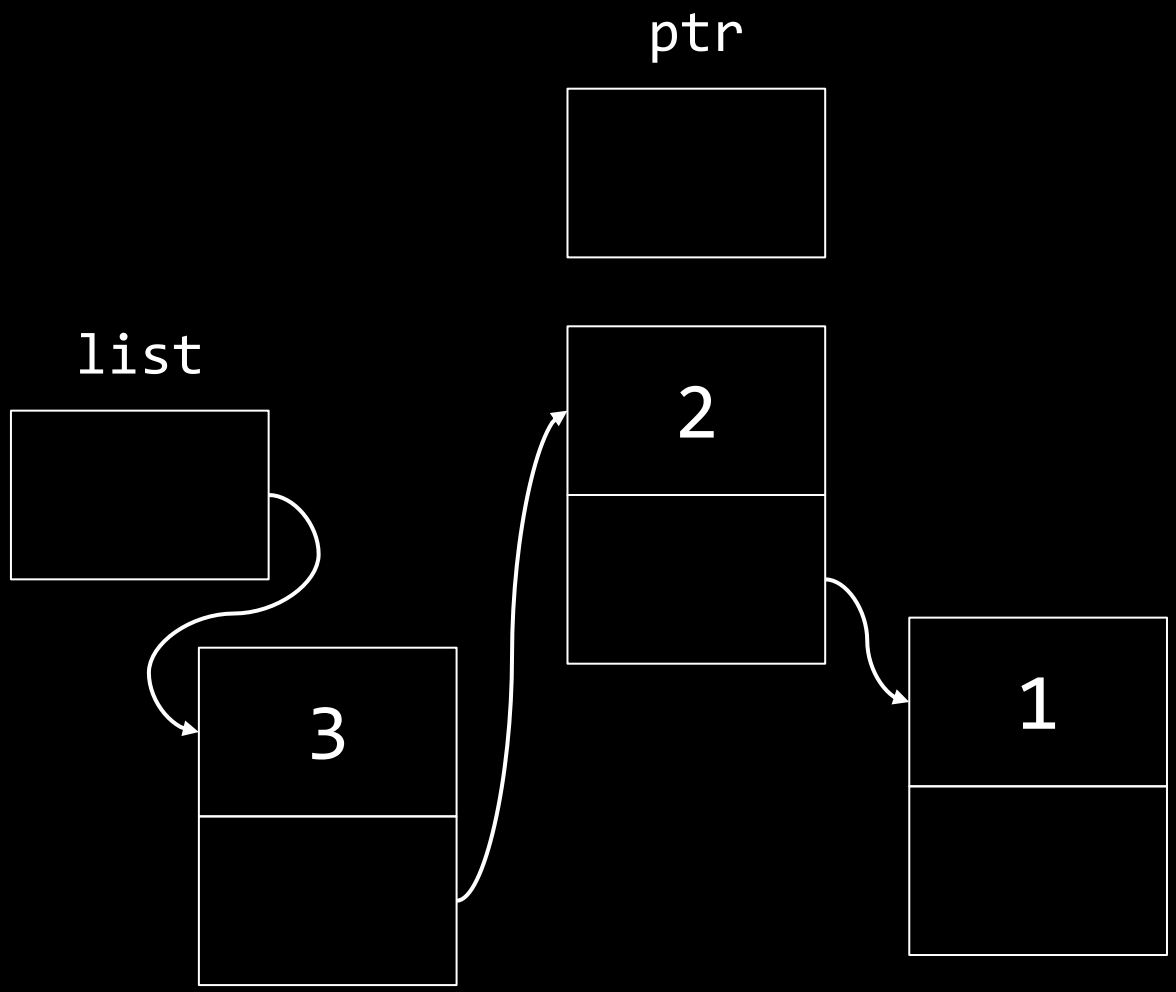

list
2
1

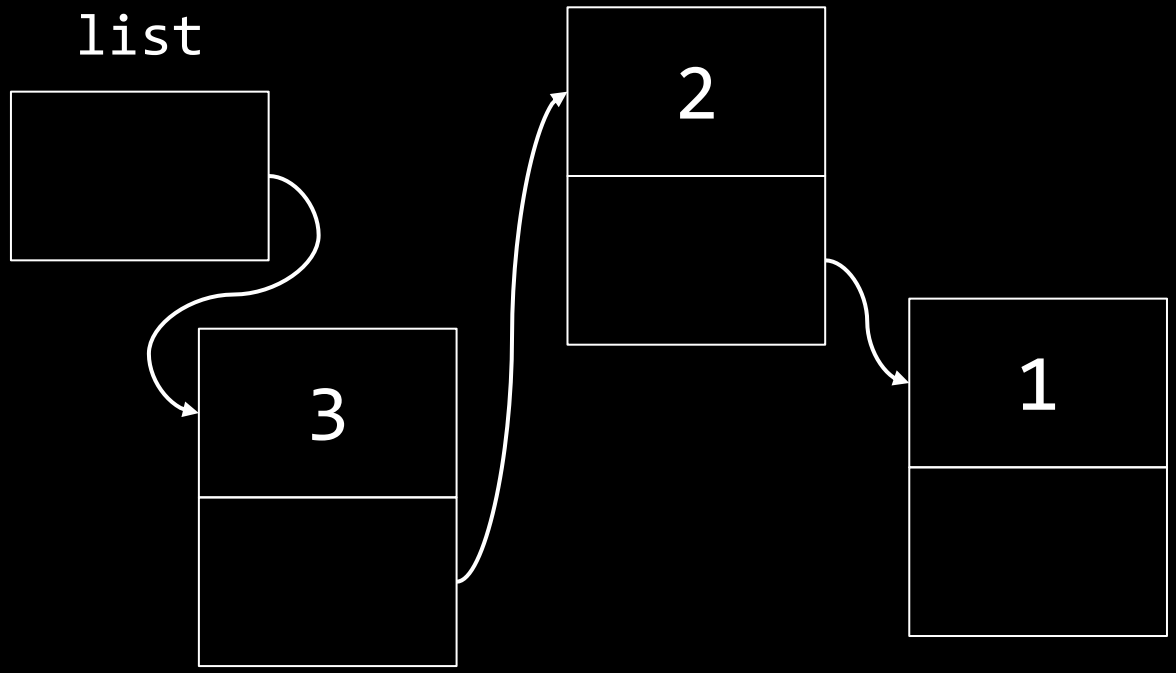

list
2
3
1


ptr
list
2
3
1


ptr
list
2
3
1


ptr
list
2
3
1


ptr
list
2
3
1


list
2
3
1

$O(n^2)$

$O(n \log n)$

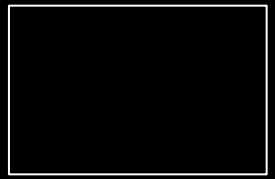
$O(n)$

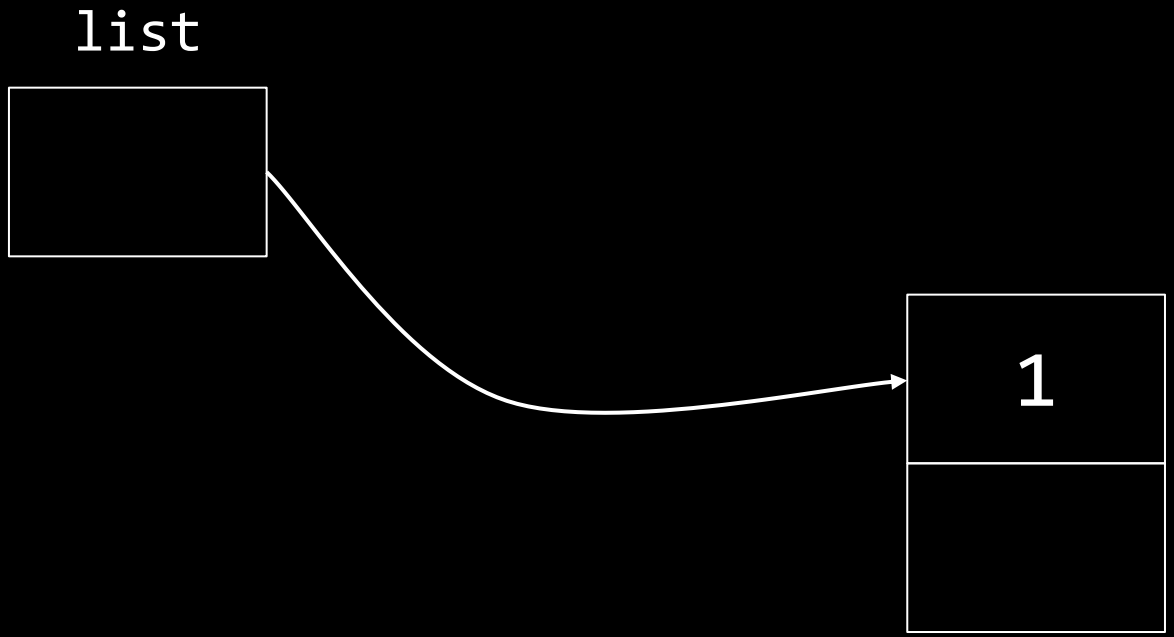
$O(\log n)$

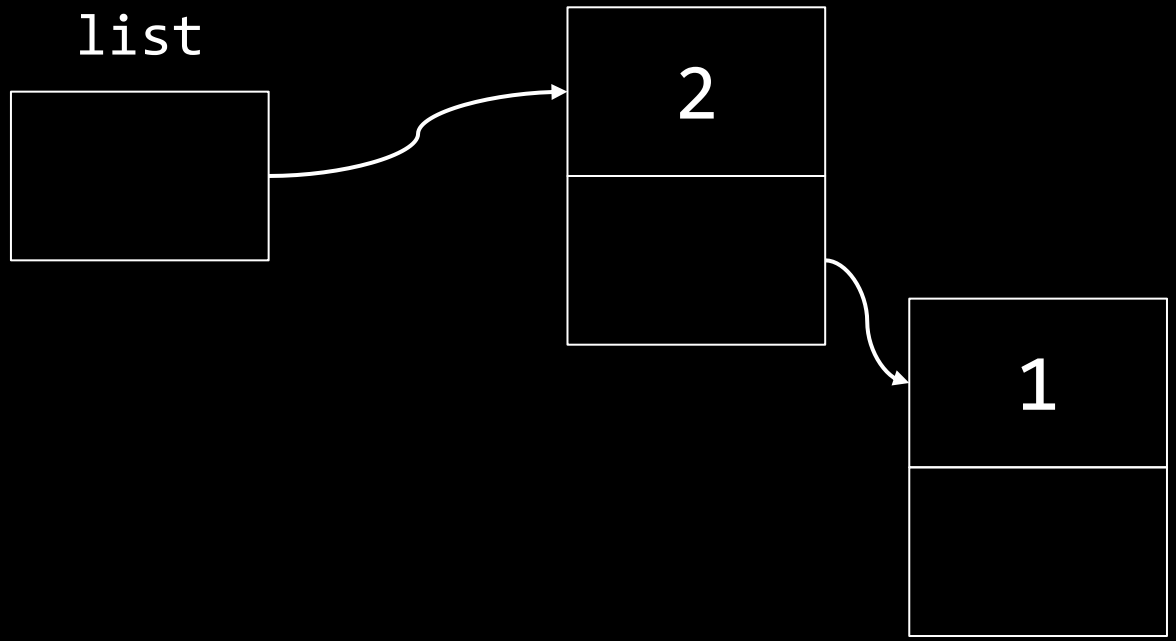
$O(1)$

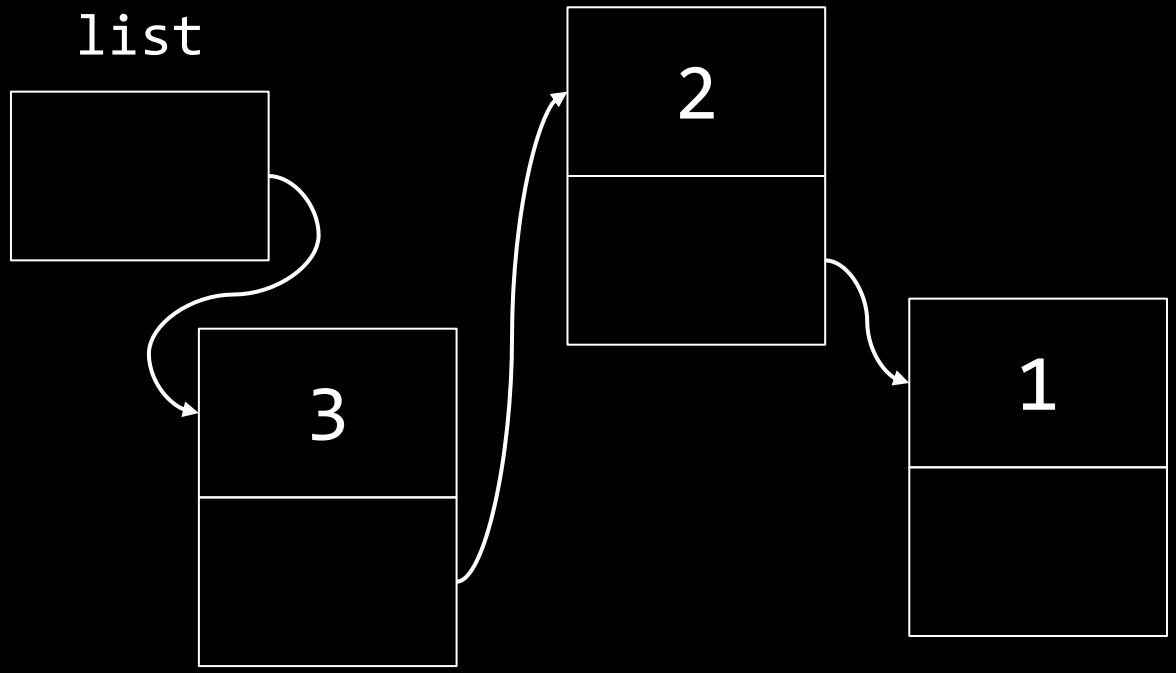
Bisher haben wir immer vorne angefügt. (prepending)

list




list
1

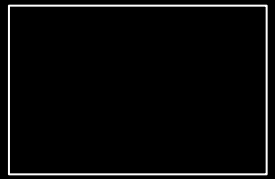

list
2
1

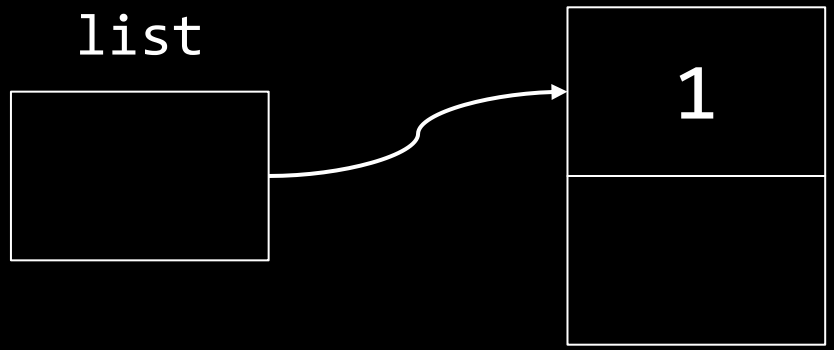

list
2
3
1

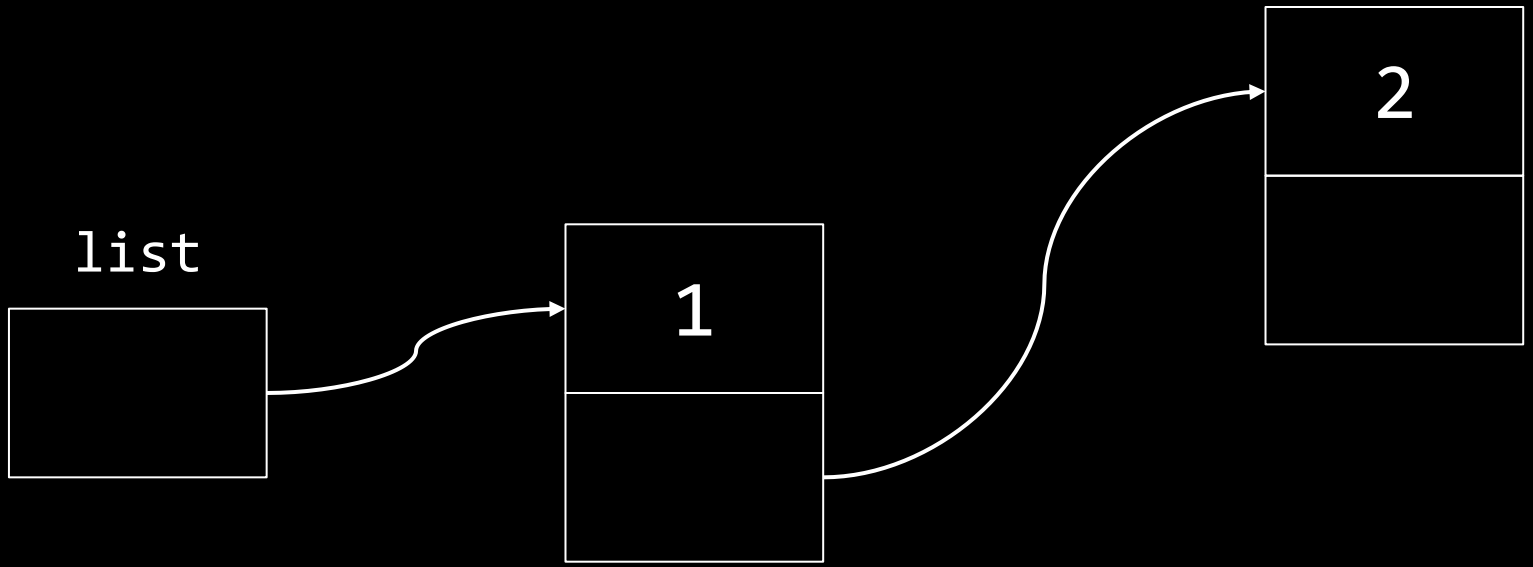
Warum?

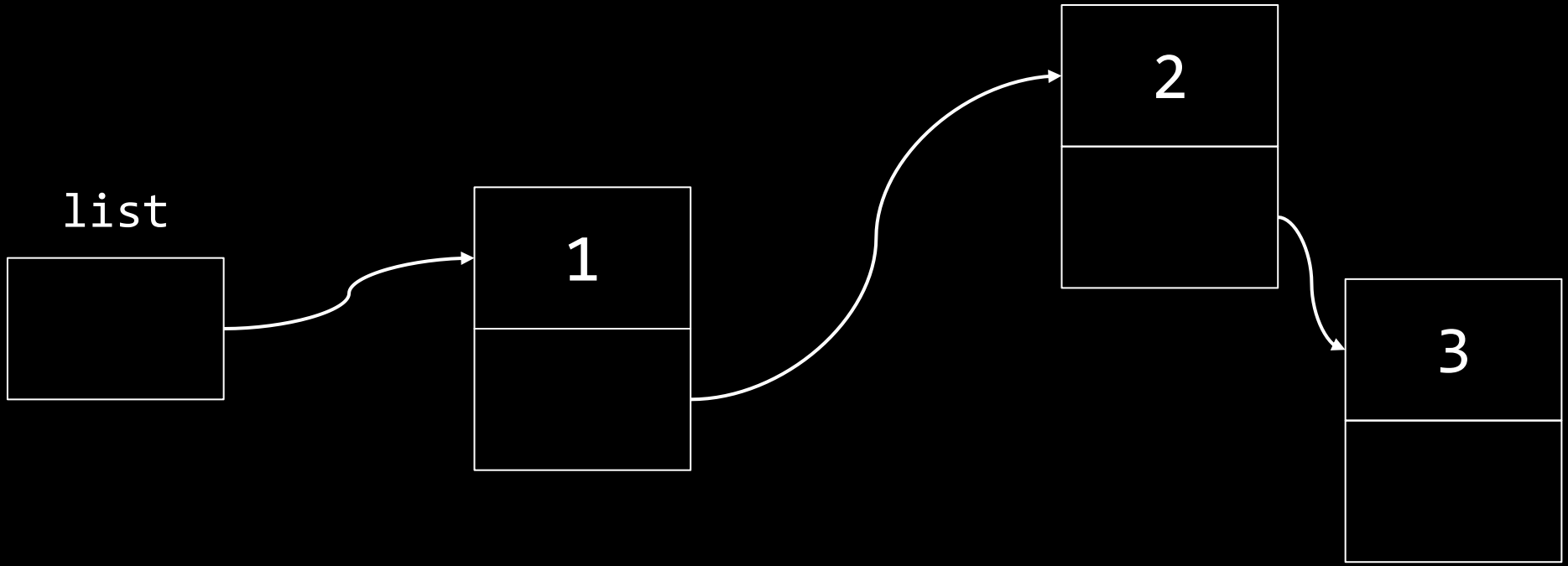
Was ist das Problem
beim Anhängen?
(appending)

list




list
1


list
1
2



siehe list5.c

$O(n^2)$

$O(n \log n)$

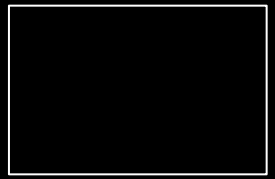
$O(n)$

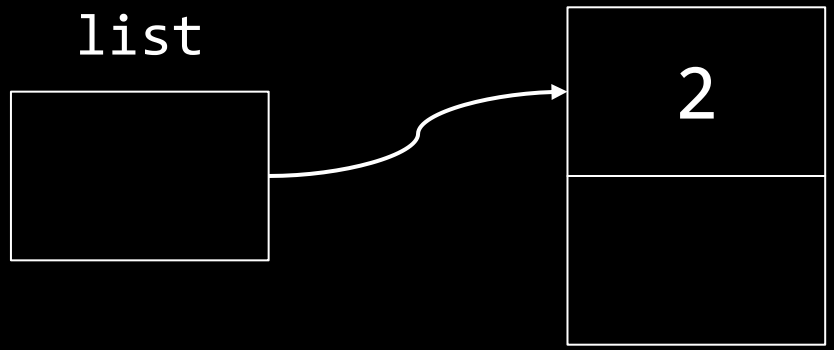
$O(\log n)$

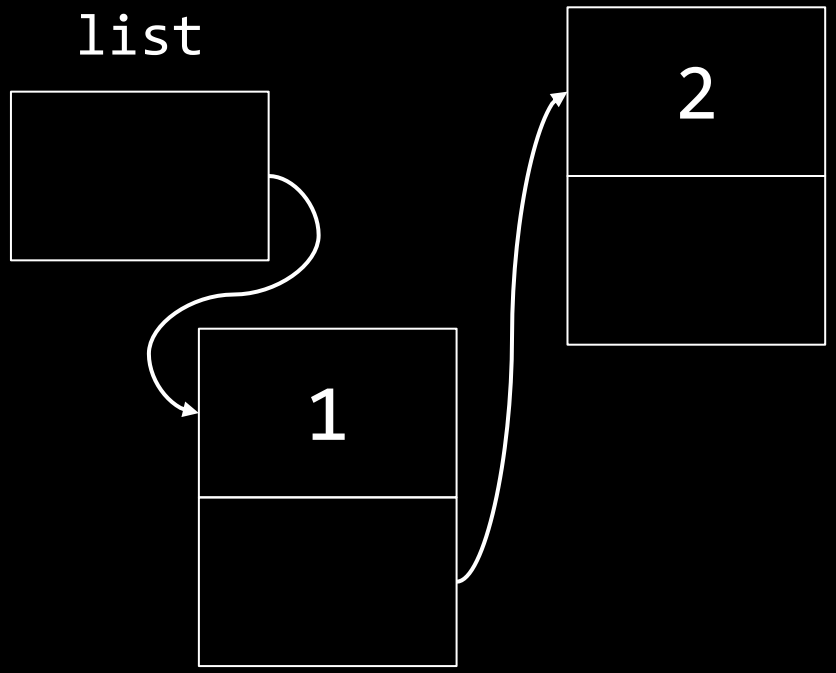
$O(1)$

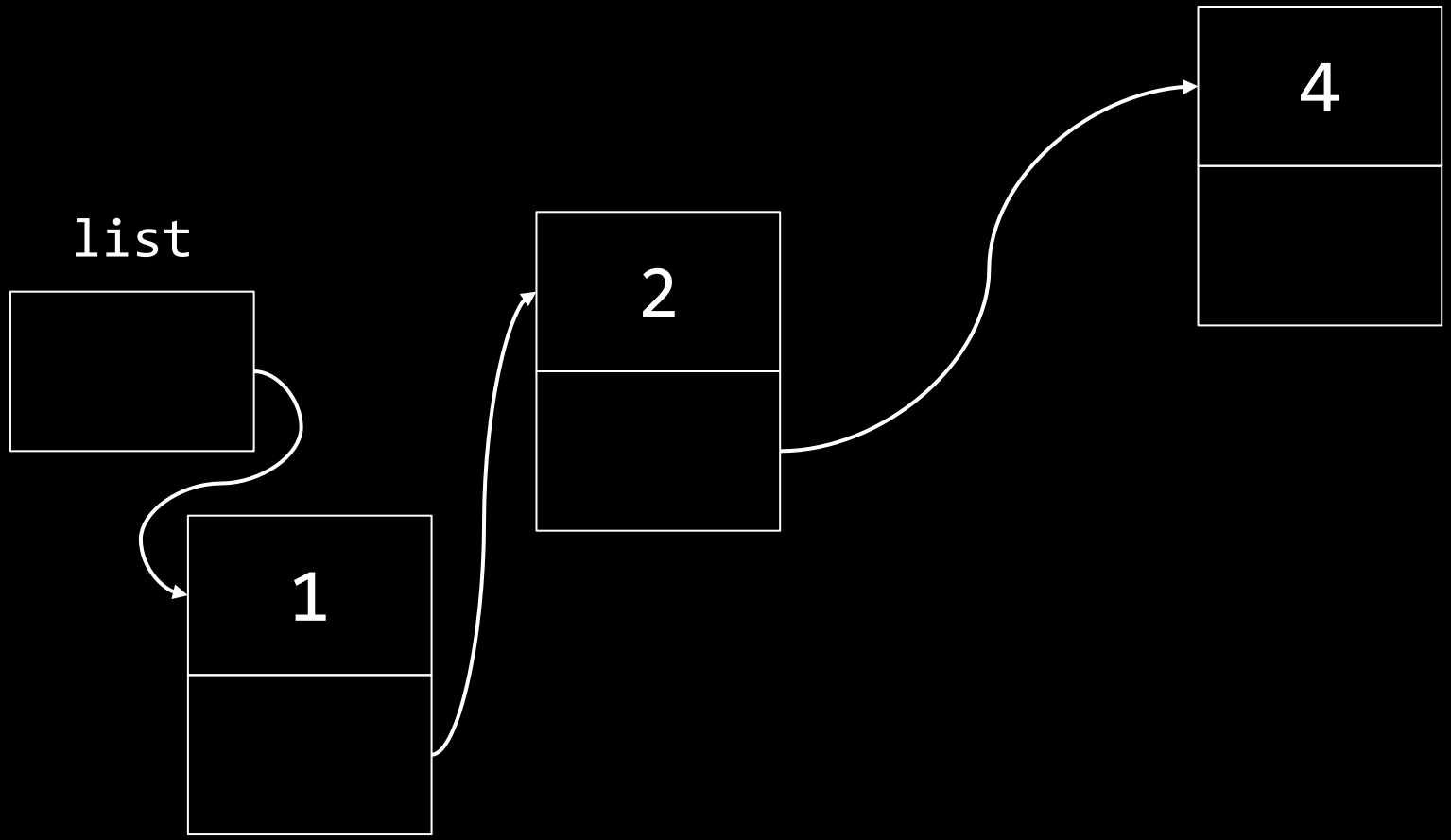
Wie könnten wir
beim Einfügen
gleich sortieren?

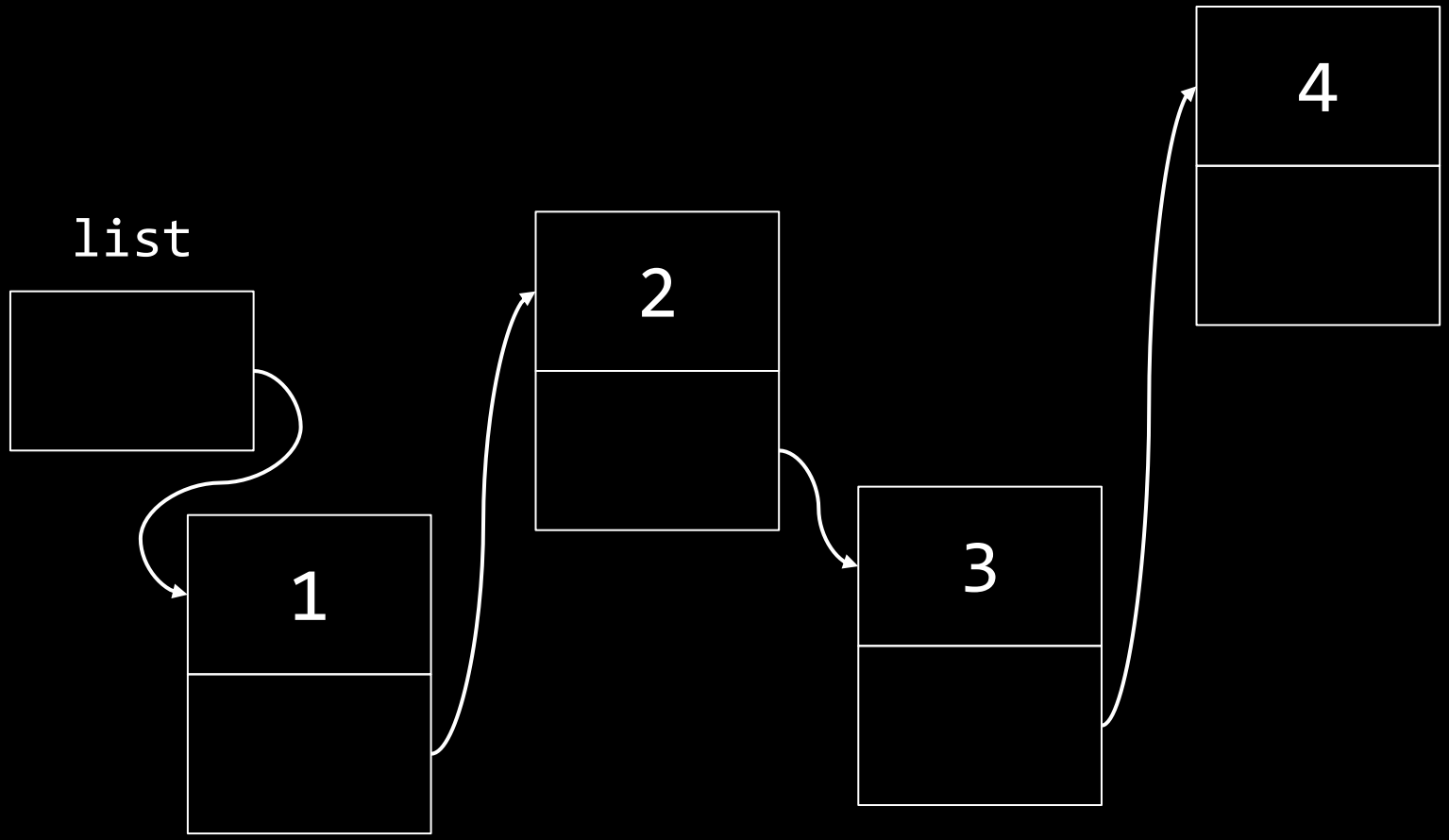
list




list
2


list
2
1


list
1
2
4


list
1
2
3
4

$O(n^2)$

$O(n \log n)$

$O(n)$

$O(\log n)$

$O(1)$

PROBLEM 3

```
node *list = NULL;
for (int i = 1; i <= 3; i++)
{
    node *n = malloc(sizeof(node));
    n->number = i;
    n->next = list;
    list = n;
}
```

Was enthält die Liste von vorne nach hinten?

PROBLEM 3

```
node *list = NULL;
for (int i = 1; i <= 3; i++)
{
    node *n = malloc(sizeof(node));
    n->number = i;
    n->next = list;
    list = n;
}
```

Lösung: 3 → 2 → 1 → NULL

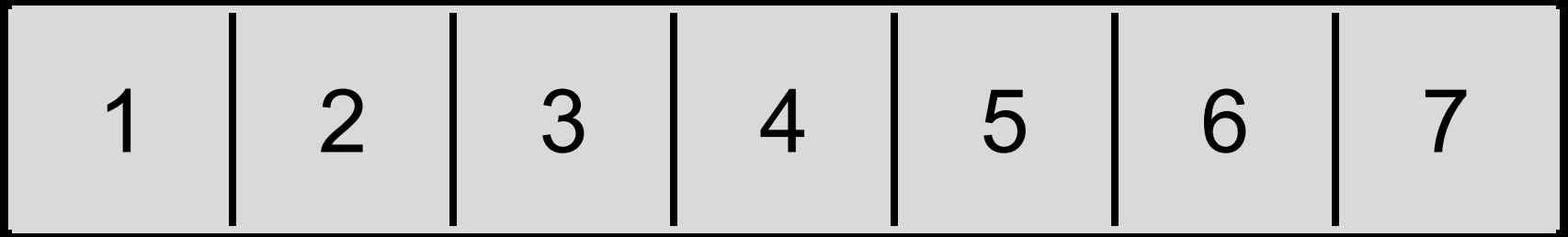
- i=1: n->number=1, n->next=NULL, list=n → Liste: 1→NULL
- i=2: n->number=2, n->next=list(→1), list=n → Liste: 2→1→NULL
- i=3: n->number=3, n->next=list(→2), list=n → Liste: 3→2→1→NULL

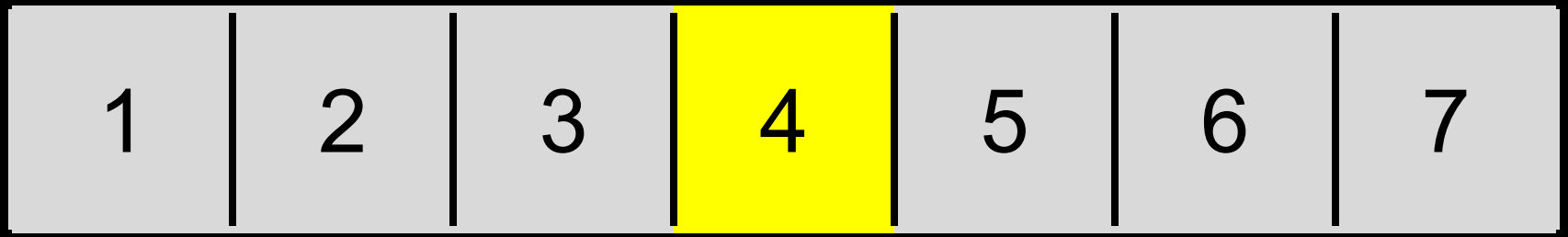
PAUSE

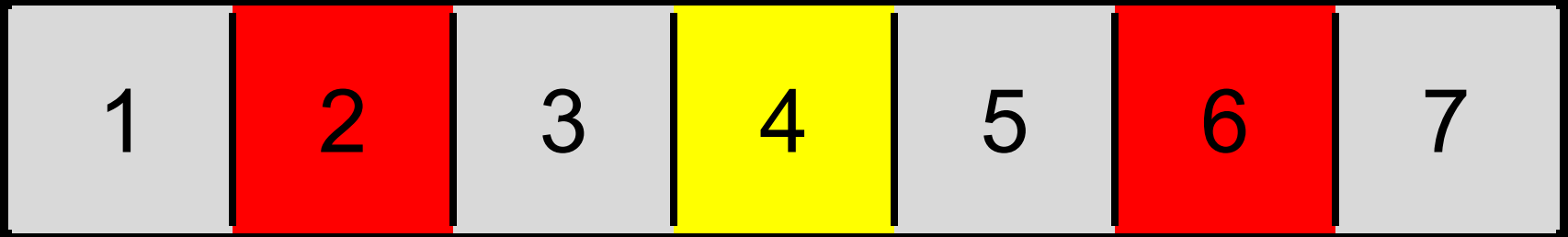
25 min

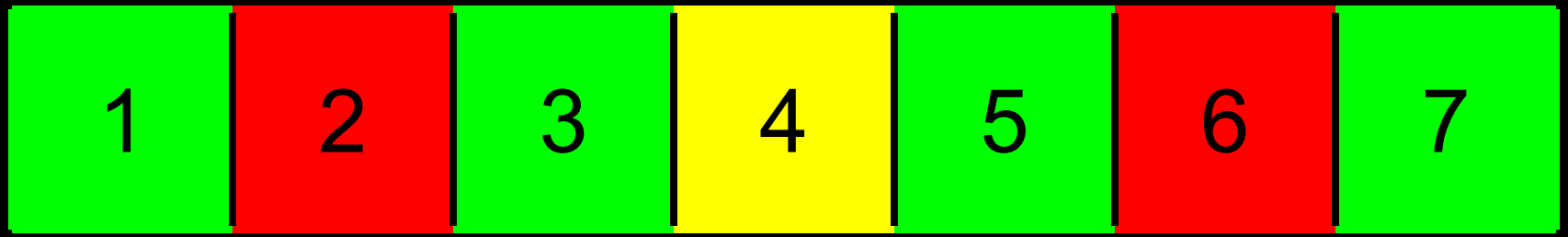
Bäume
(Trees)

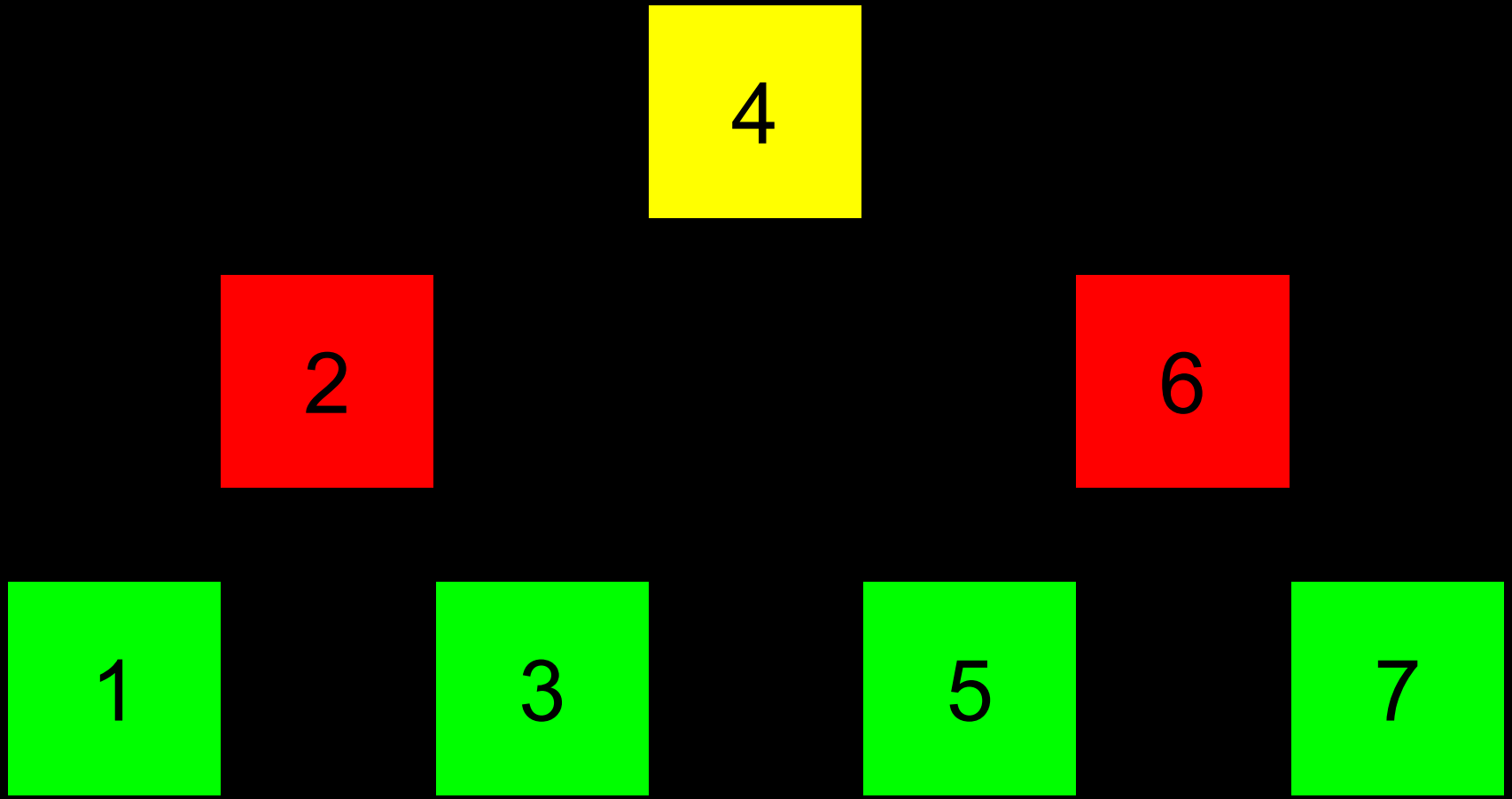
Binäre Suchbäume
(Binary Search Trees)

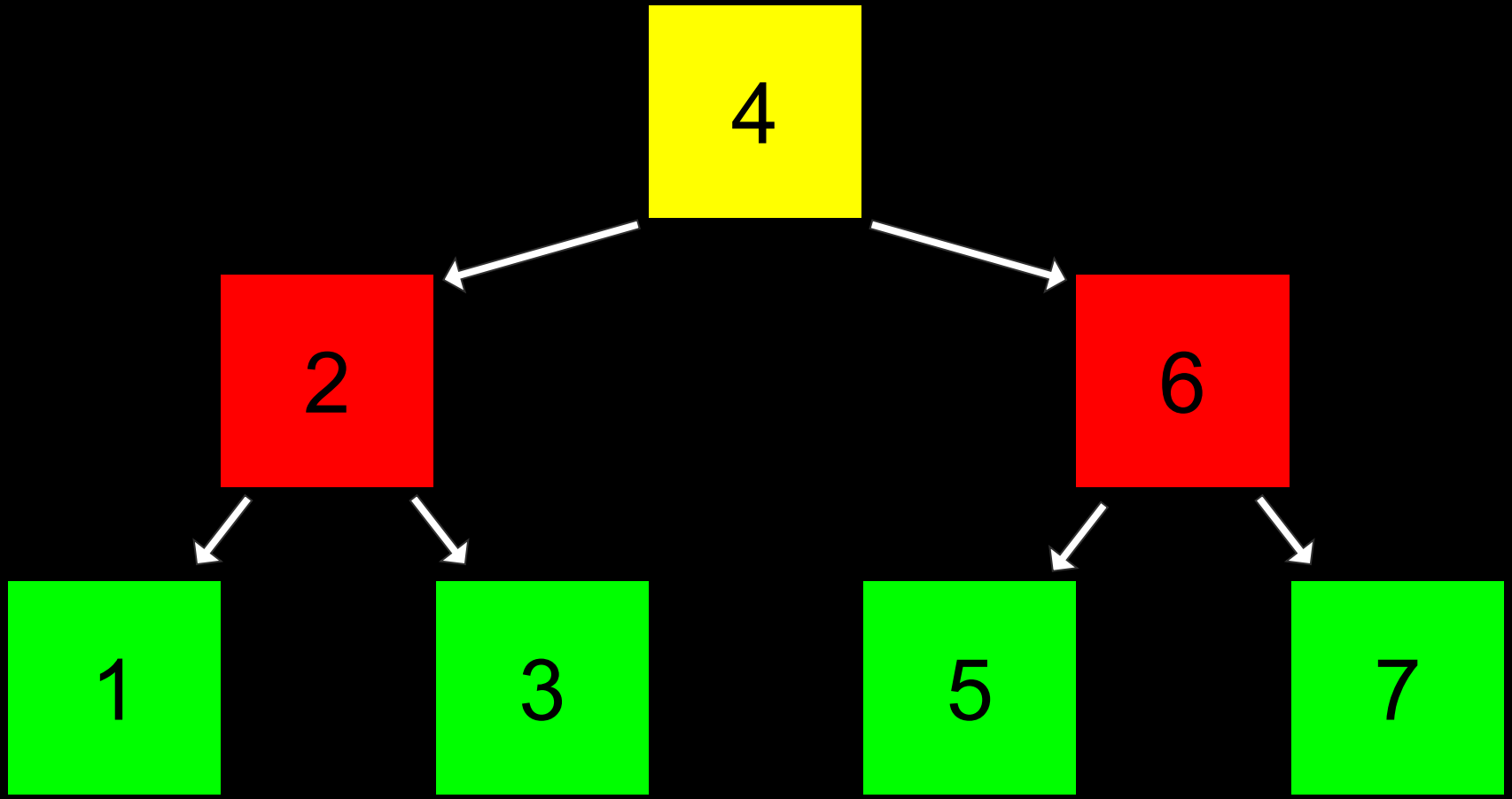

1
2
3
4
5
6
7


1
2
3
4
5
6
7


1
2
3
4
5
6
7


1
2
3
4
5
6
7


4
2
6
1
3
5
7

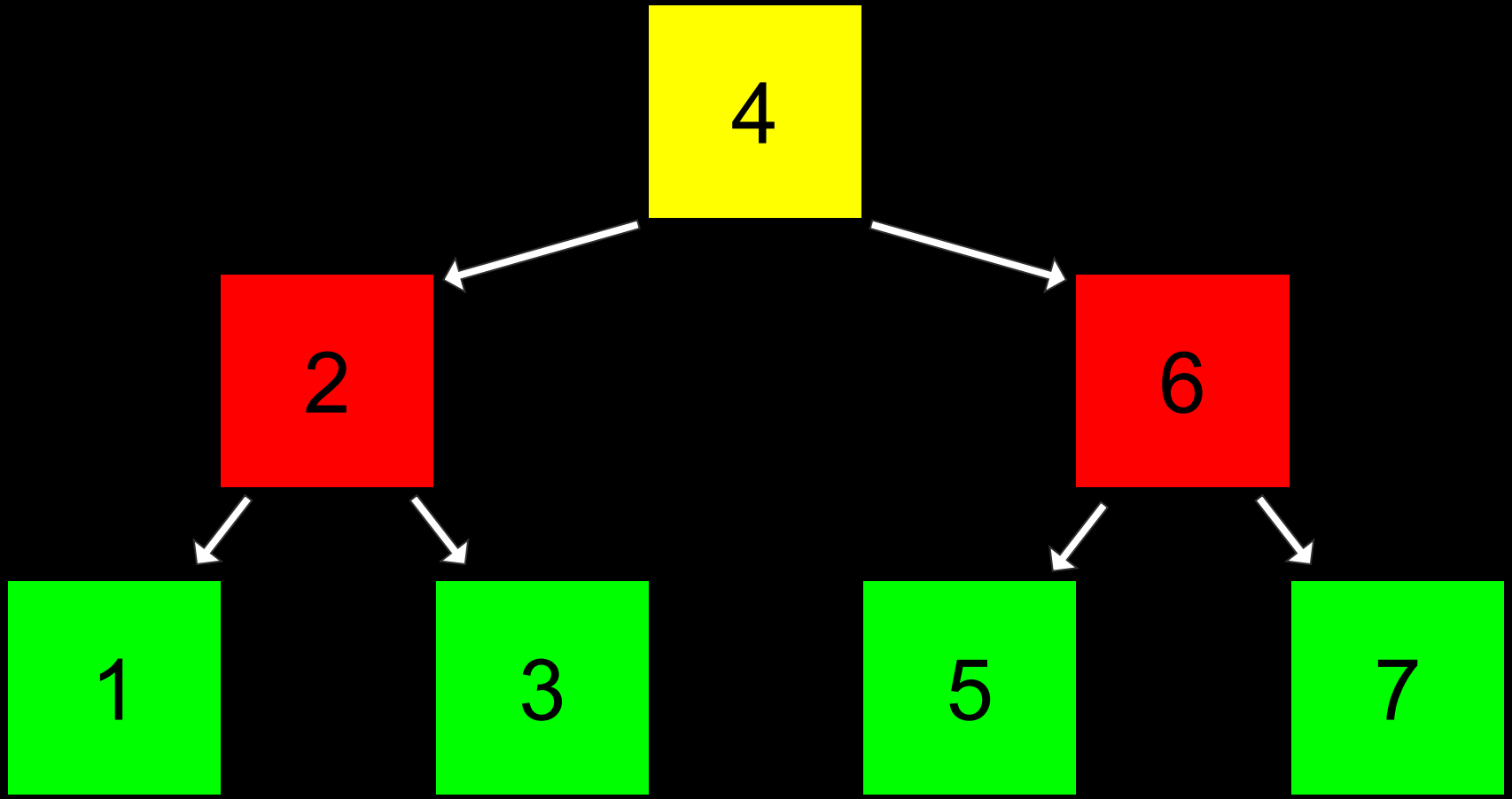

4
2
6
1
3
5
7

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

```
typedef struct node
{
    int number;

} node;
```

```
typedef struct node
{
    int number;


} node;
```

```
typedef struct node
{
    int number;
    struct node *left;
    struct node *right;
} node;
```


4
2
6
1
3
5
7

```
bool search(node *tree, int number)
{









}
```

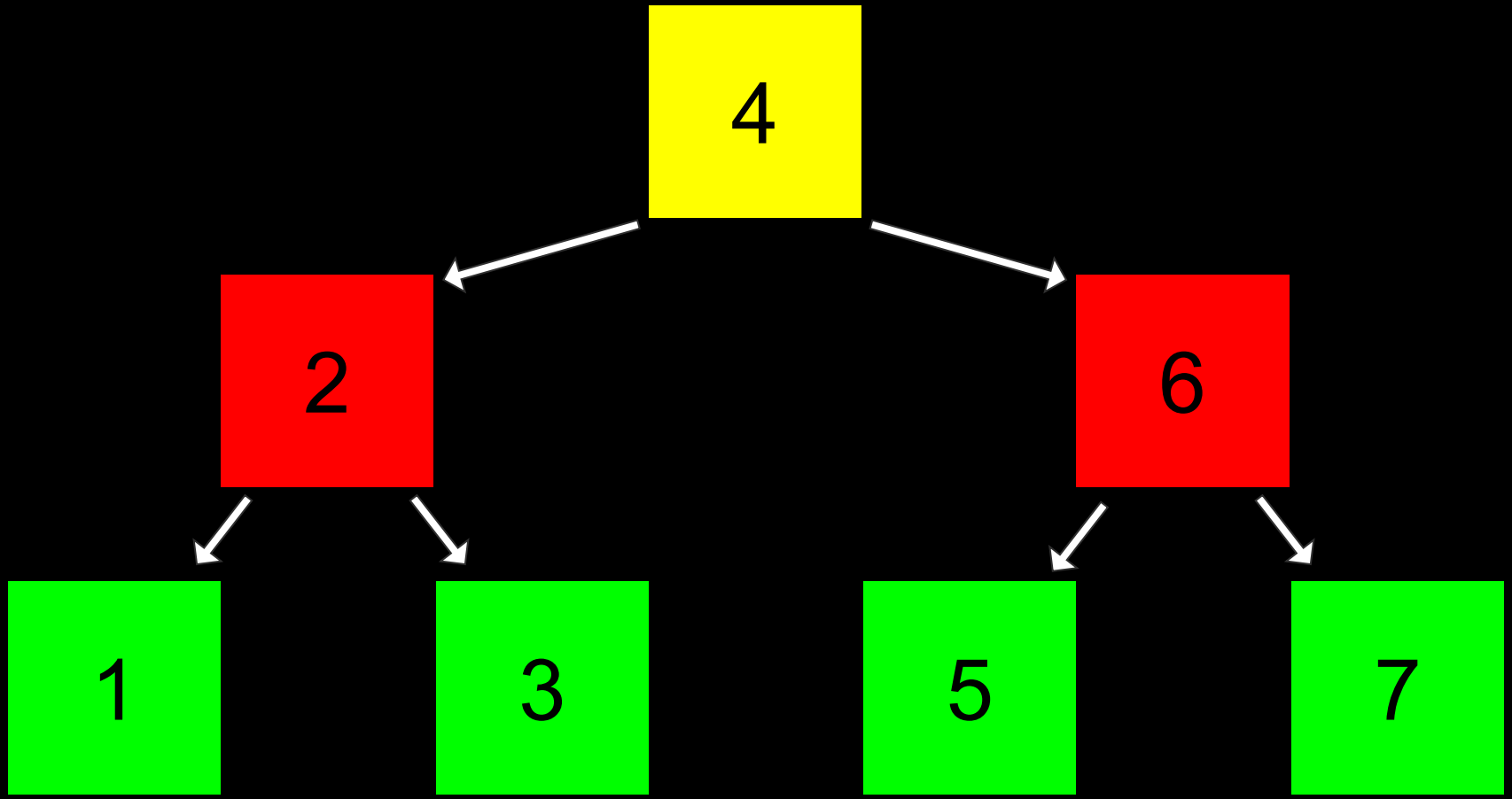
```
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }









}
```

```
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }









}
```

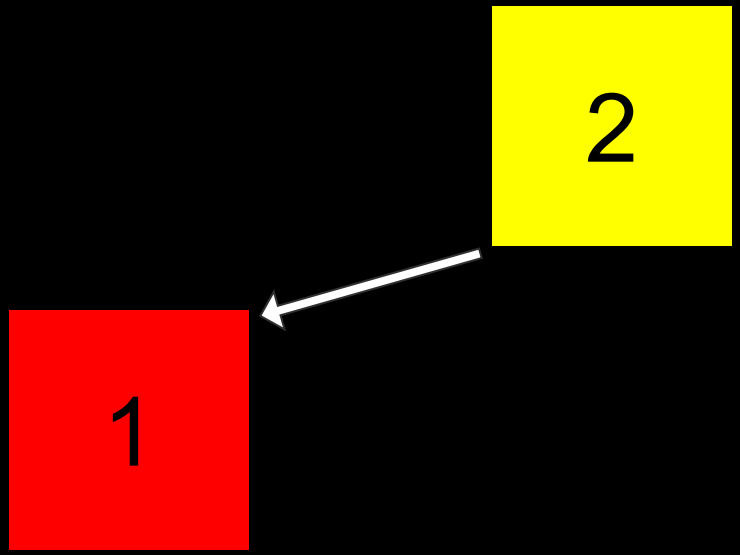
```
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }




}
```

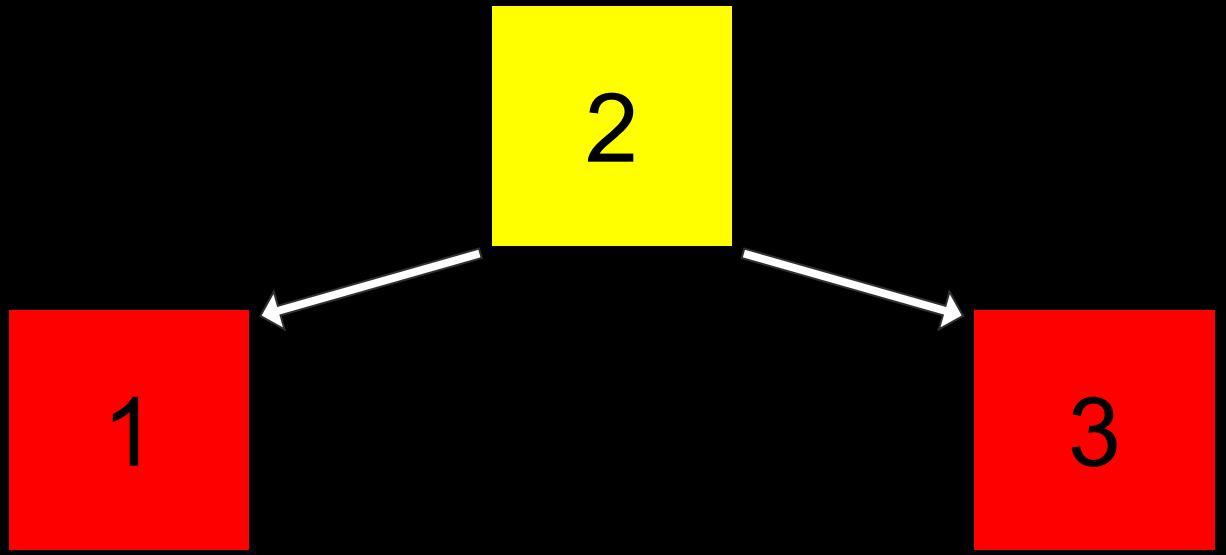
```
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else if (number == tree->number)
    {
        return true;
    }
}
```

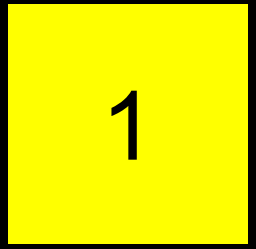
```
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else
    {
        return true;
    }
}
```

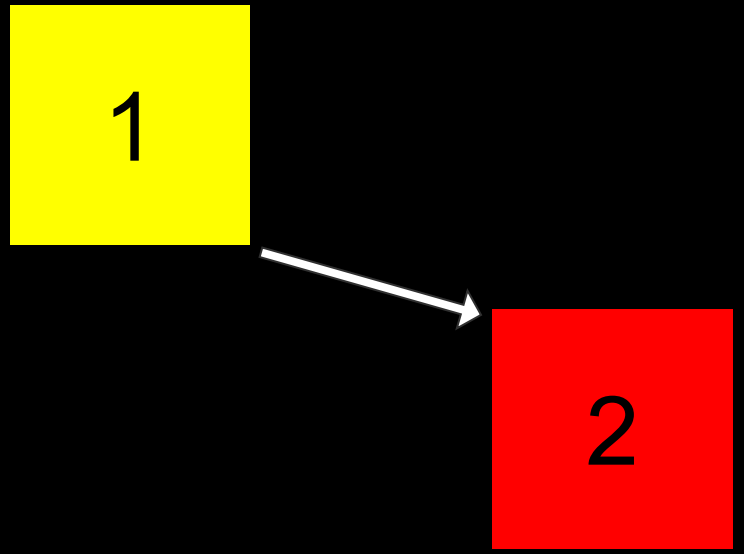

4
2
6
1
3
5
7

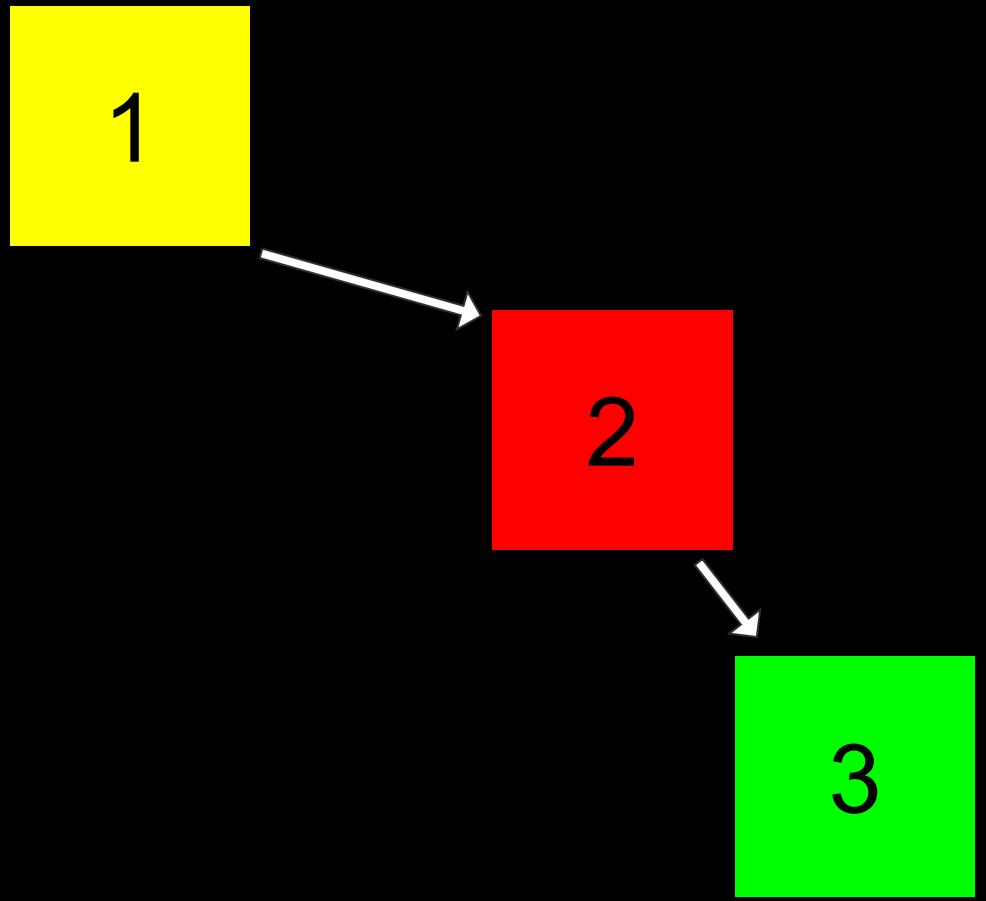
2


2
1


2
1
3


1


1
2


1
2
3

$O(n^2)$

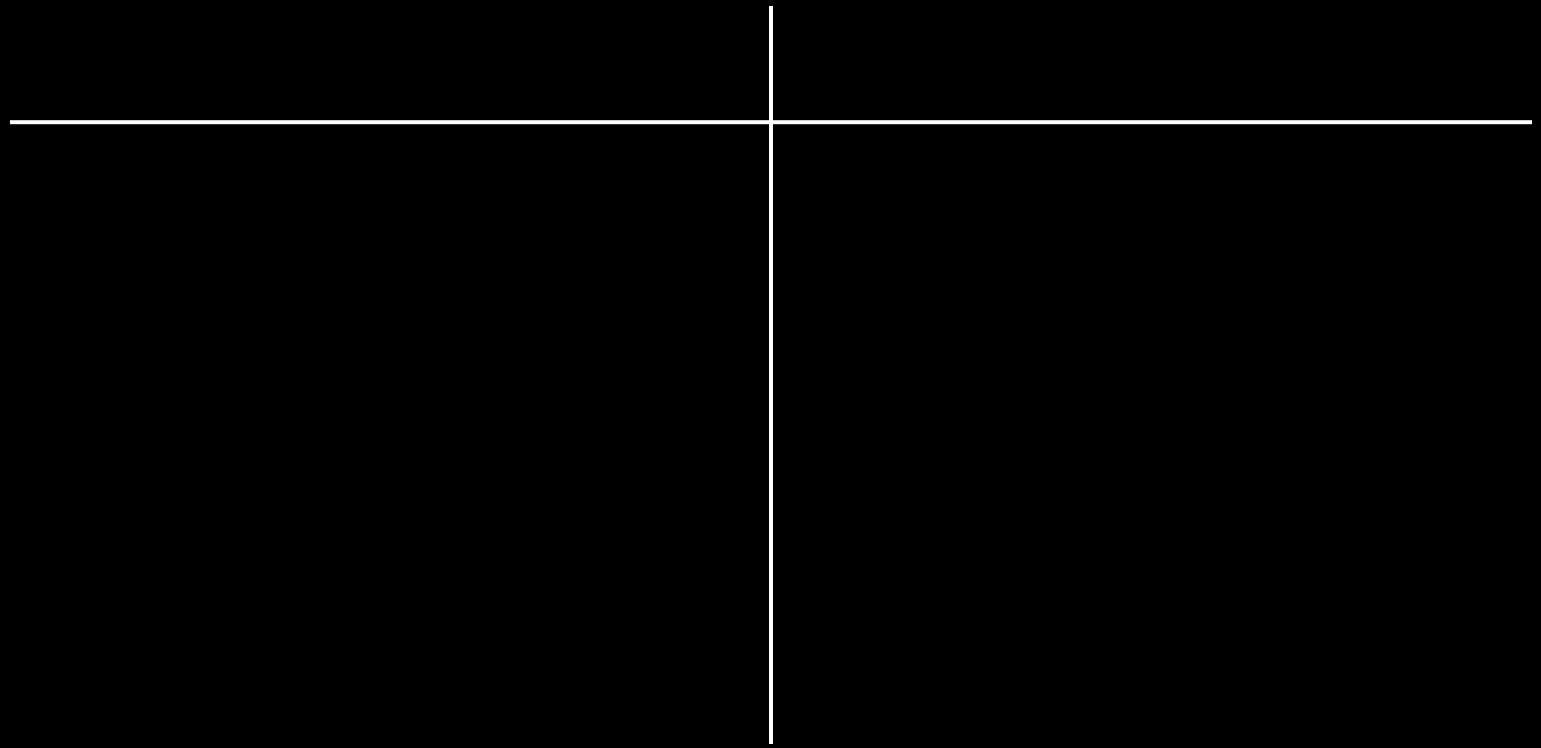
$O(n \log n)$

$O(n)$

$O(\log n)$

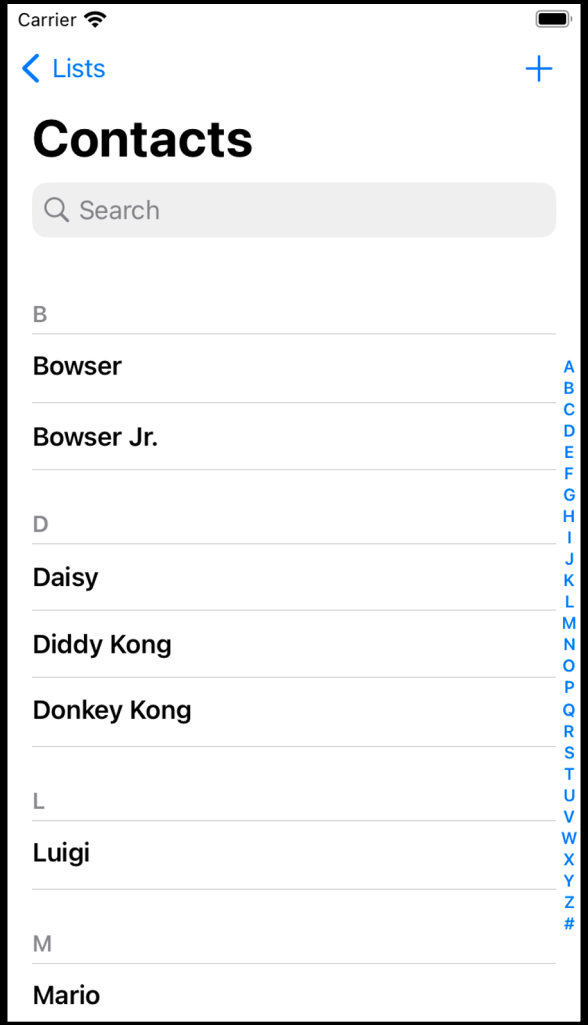
$O(1)$

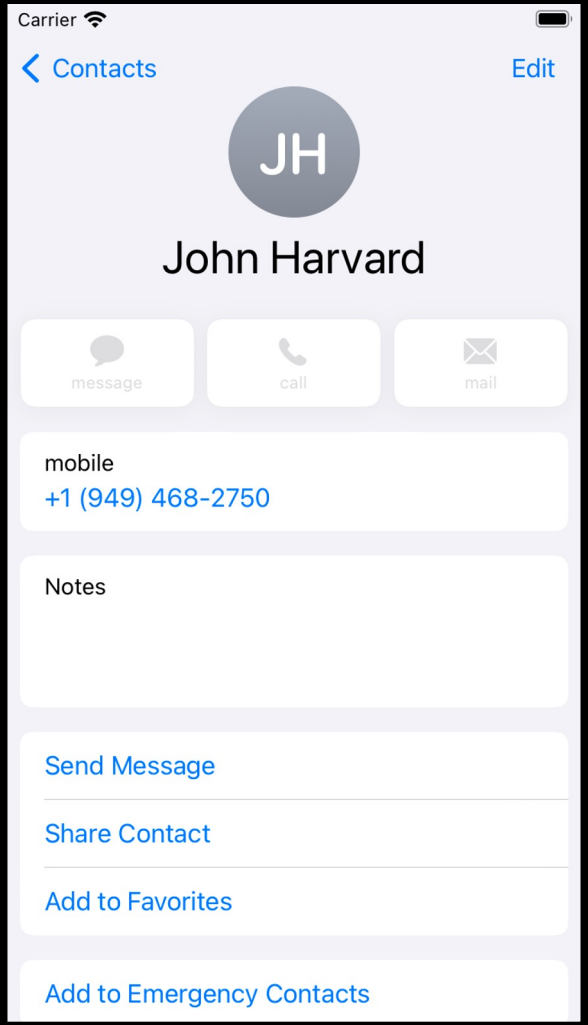
Dictionaries
(Wörterbücher)



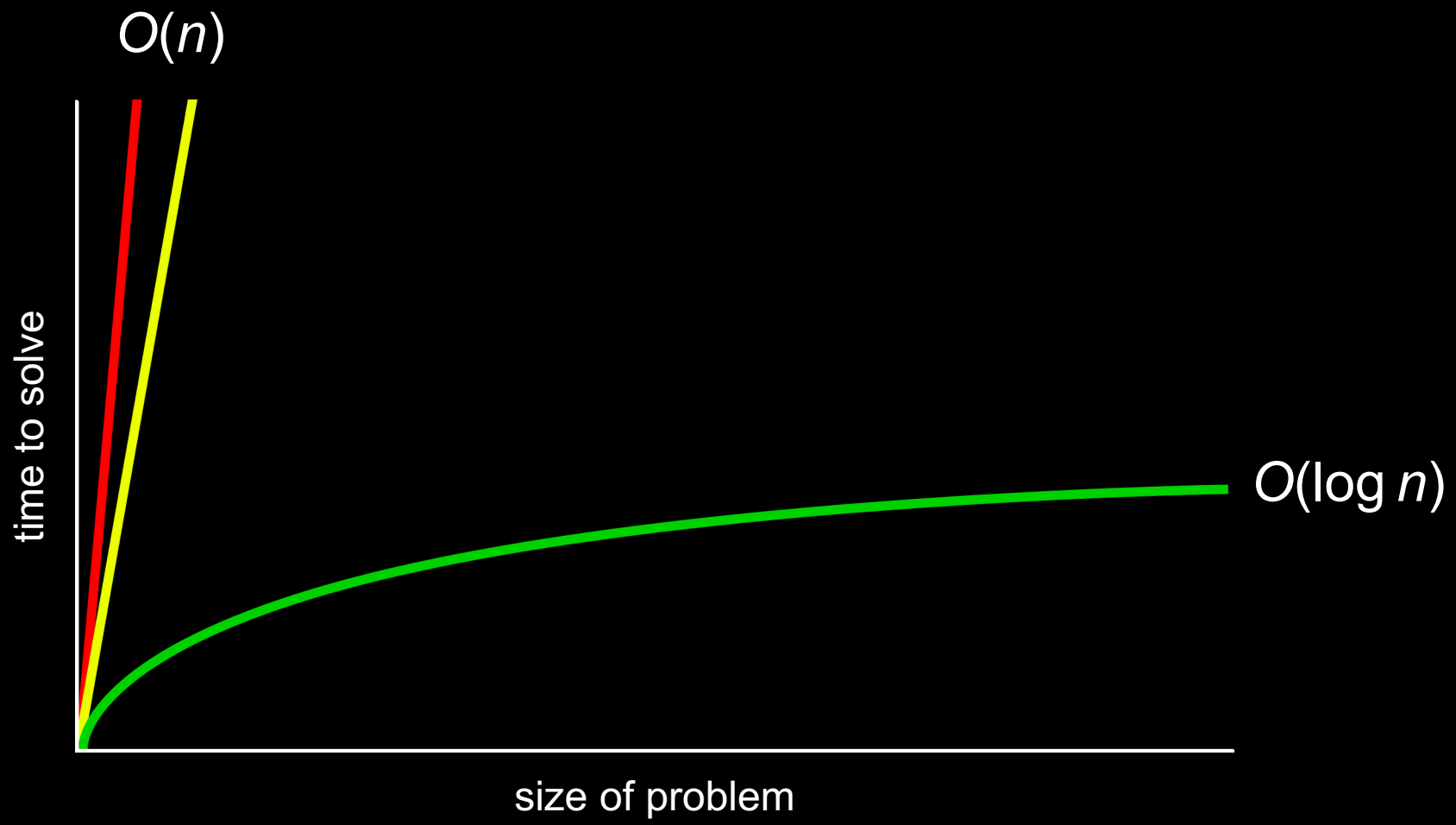
Wort	Definition

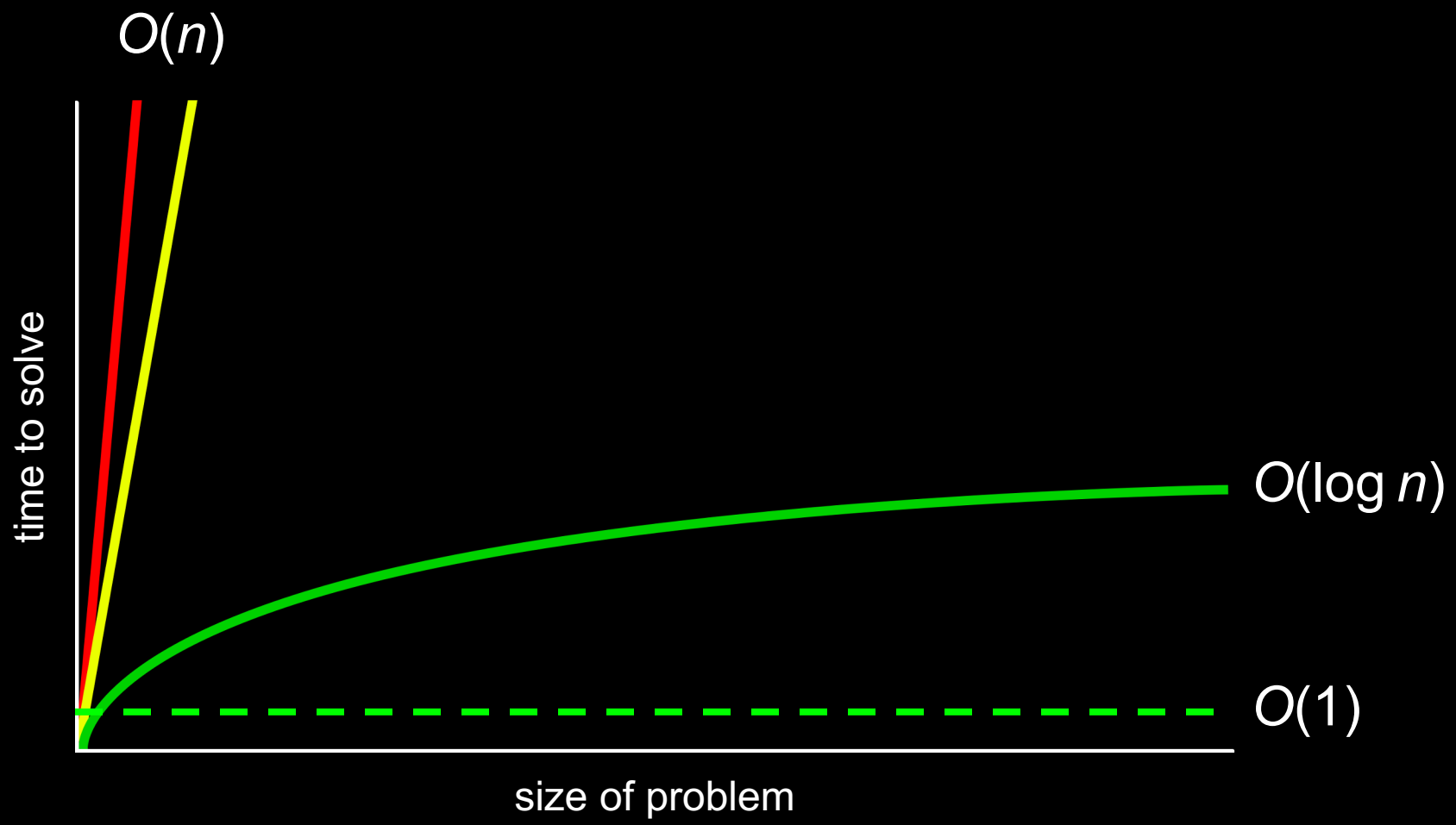
Key	Value


Carrier
Lists
Contacts
Search
B
Bowser
Bowser Jr.
D
Daisy
Diddy Kong
Donkey Kong
L
Luigi
M
Mario
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
#


Carrier
Contacts
Edit
JH
John Harvard
message
call
mail
mobile
+1 (949) 468-2750
Notes
Send Message
Share Contact
Add to Favorites
Add to Emergency Contacts

Name	Nummer

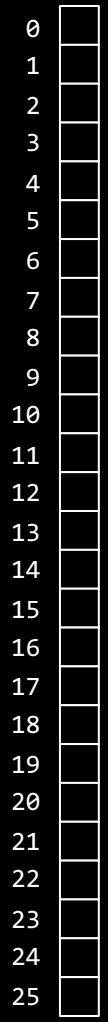

O(n)
time to solve
O(log n)
size of problem


O(n)
time to solve
O(log n)
O(1)
size of problem

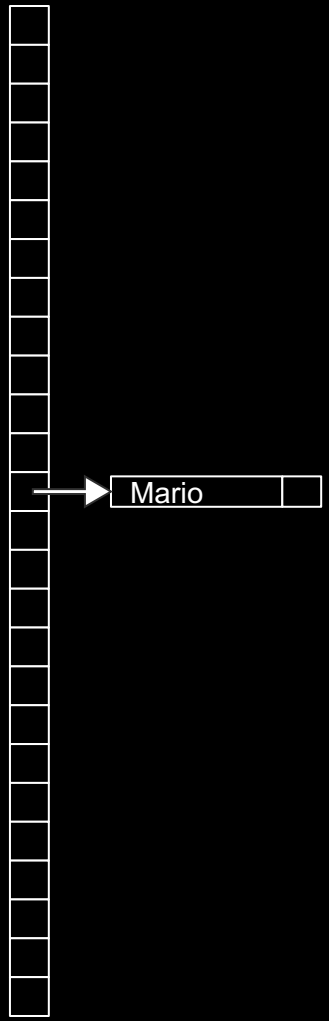
Hashing

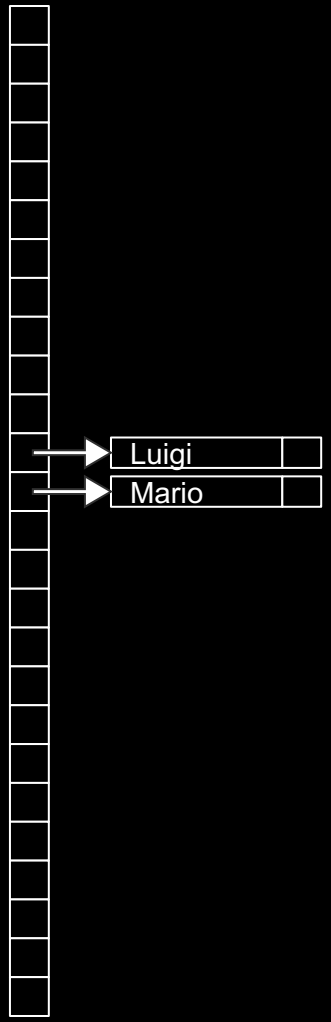
Hash-Funktionen

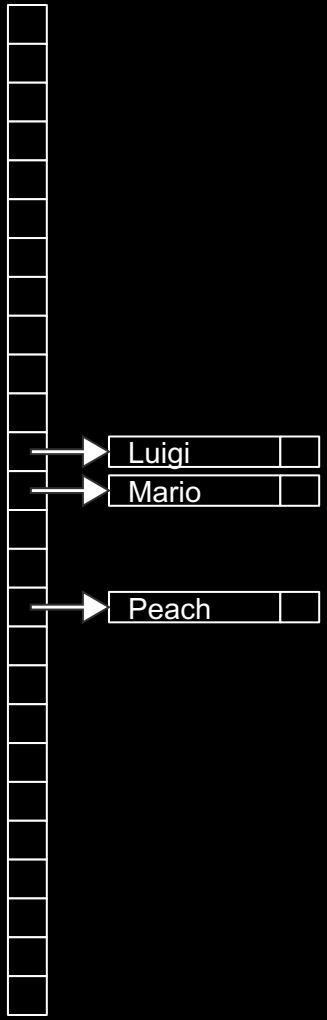
Hash-Tabellen

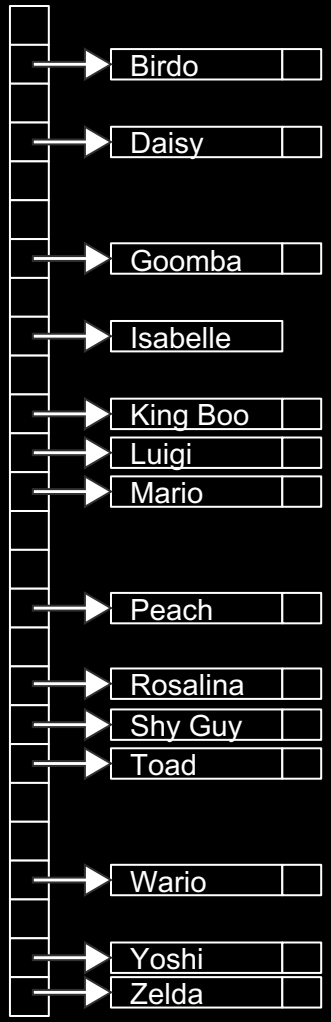

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

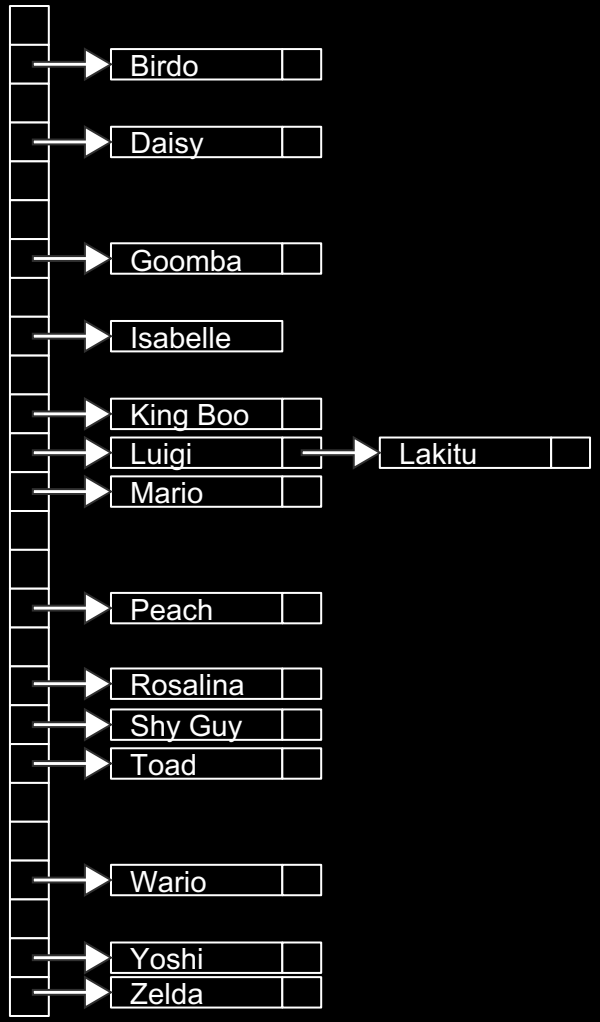

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z

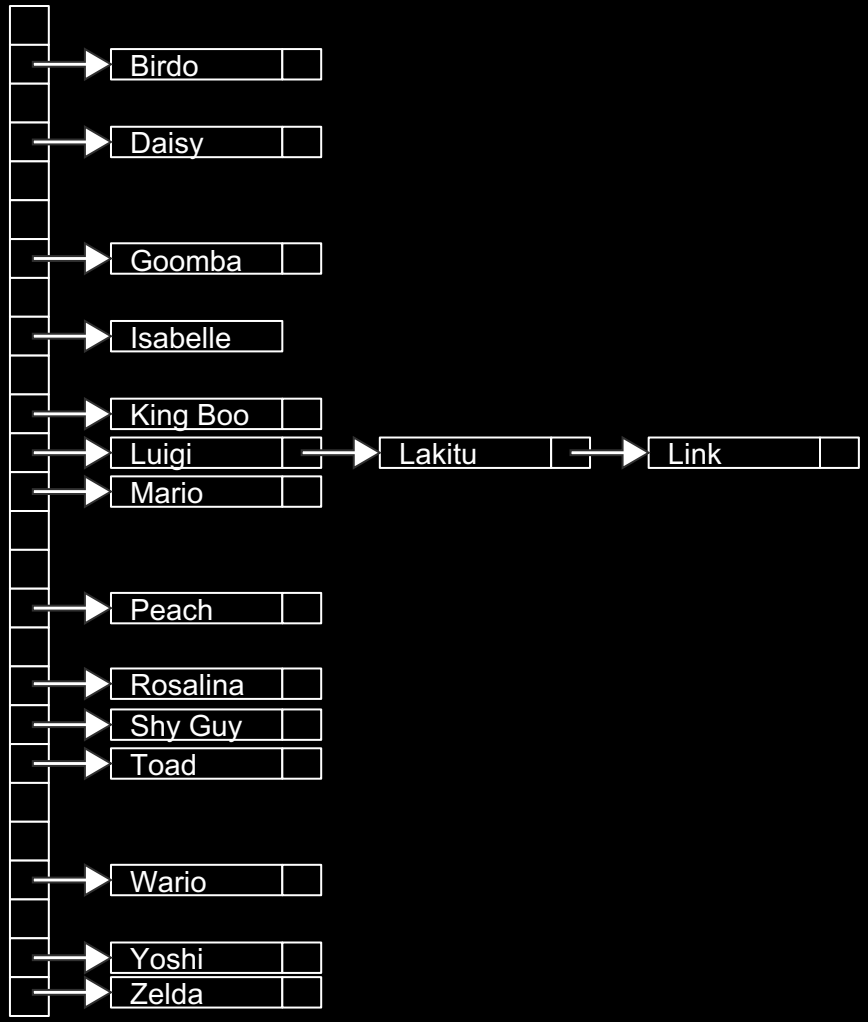

Mario

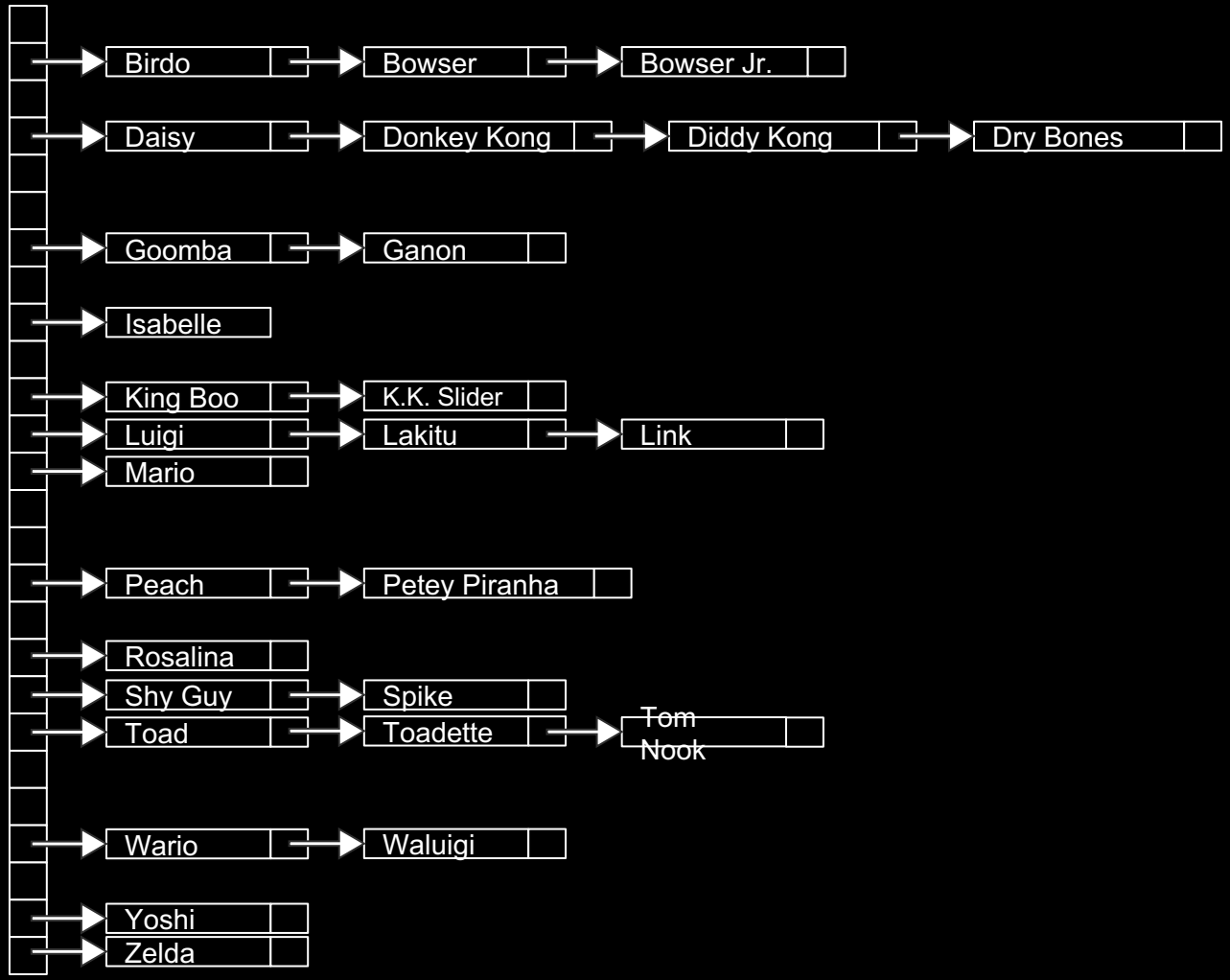

Luigi
Mario


Luigi
Mario
Peach


Birdo
Daisy
Goomba
Isabelle
King Boo
Luigi
Mario
Peach
Rosalina
Shy Guy
Toad
Wario
Yoshi
Zelda


Birdo
Daisy
Goomba
Isabelle
King Boo
Luigi
Lakitu
Mario
Peach
Rosalina
Shy Guy
Toad
Wario
Yoshi
Zelda


Birdo
Daisy
Goomba
Isabelle
King Boo
Luigi
Lakitu
Link
Mario
Peach
Rosalina
Shy Guy
Toad
Wario
Yoshi
Zelda


Birdo
Bowser
Bowser Jr.
Daisy
Donkey Kong
Diddy Kong
Dry Bones
Goomba
Ganon
Isabelle
King Boo
K.K. Slider
Luigi
Lakitu
Link
Mario
Peach
Petey Piranha
Rosalina
Shy Guy
Spike
Toad
Toadette
Tom Nook
Wario
Waluigi
Yoshi
Zelda

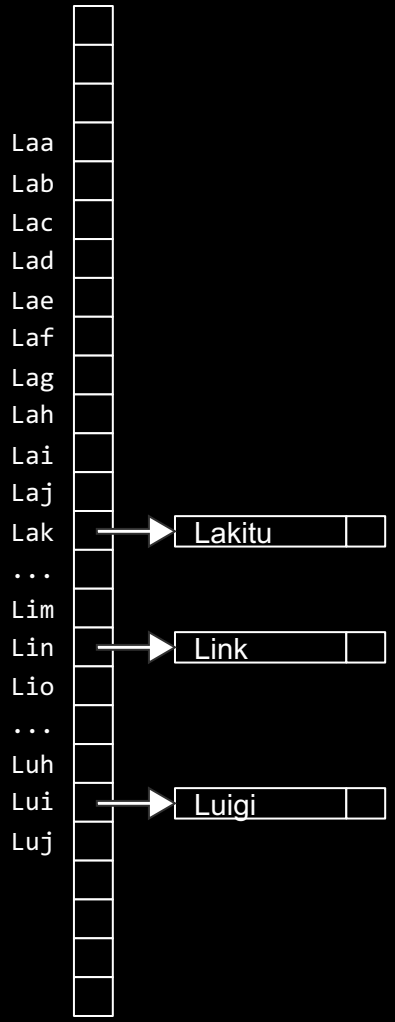
$O(n^2)$

$O(n \log n)$

$O(n)$

$O(\log n)$

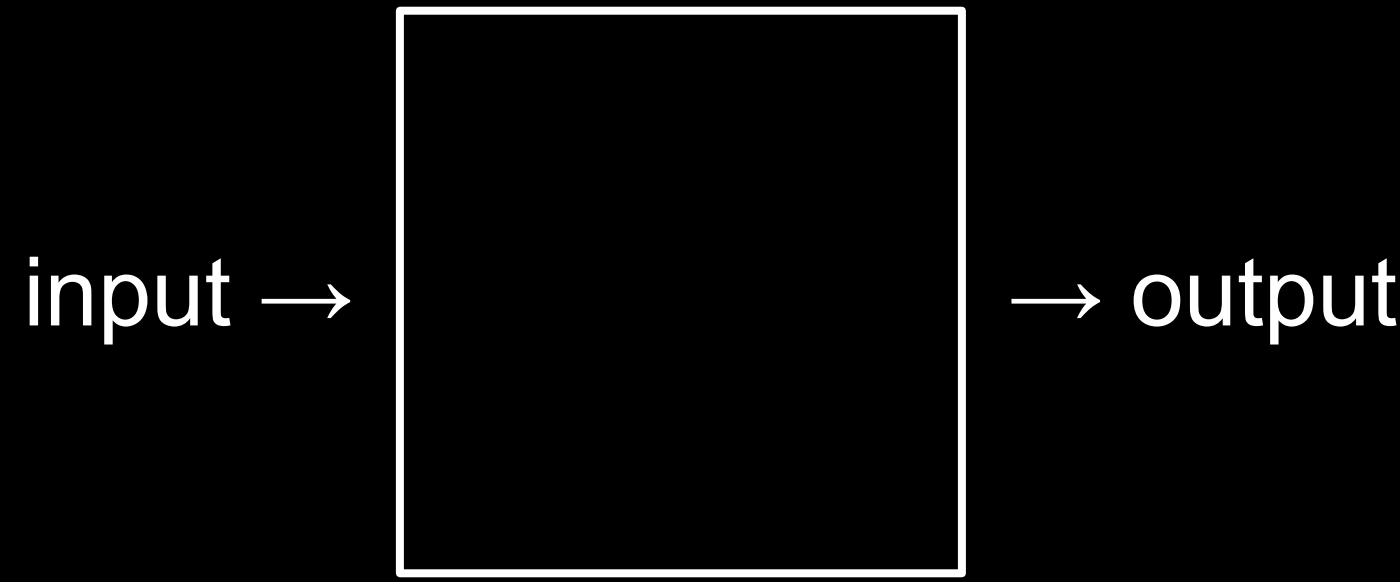
$O(1)$


Laa
Lab
Lac
Lad
Lae
Laf
Lag
Lah
Lai
Laj
Lak
...
Lim
Lin
Lio
...
Luh
Lui
Luj
Lakitu
Link
Luigi

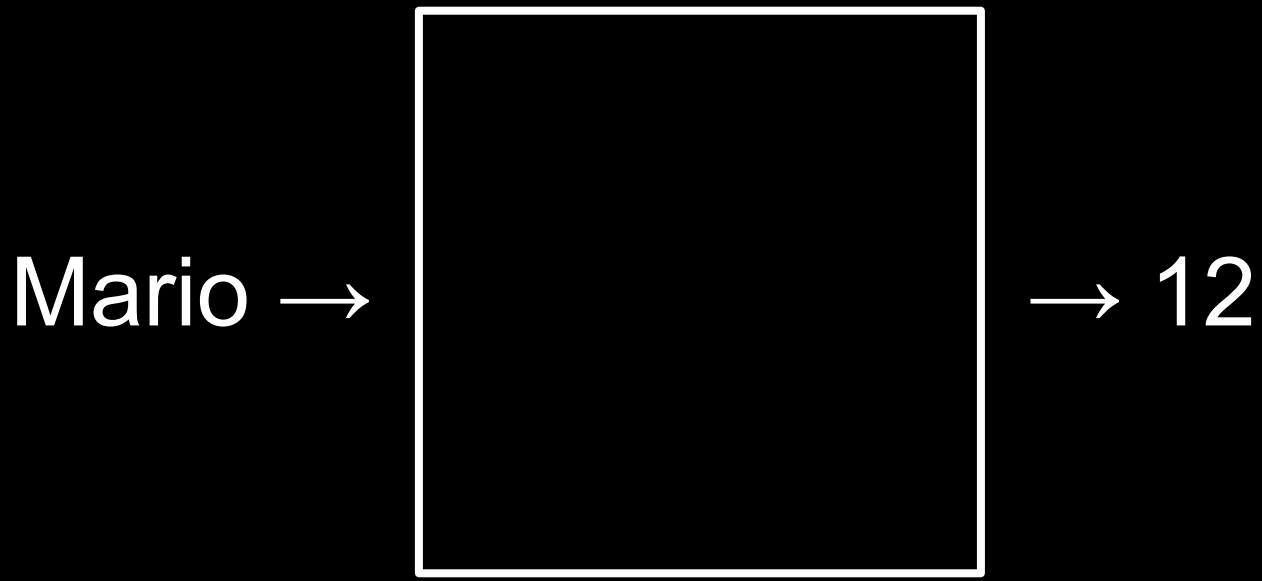
```
typedef struct
{
    char *name;
    char *number;
} person;
```

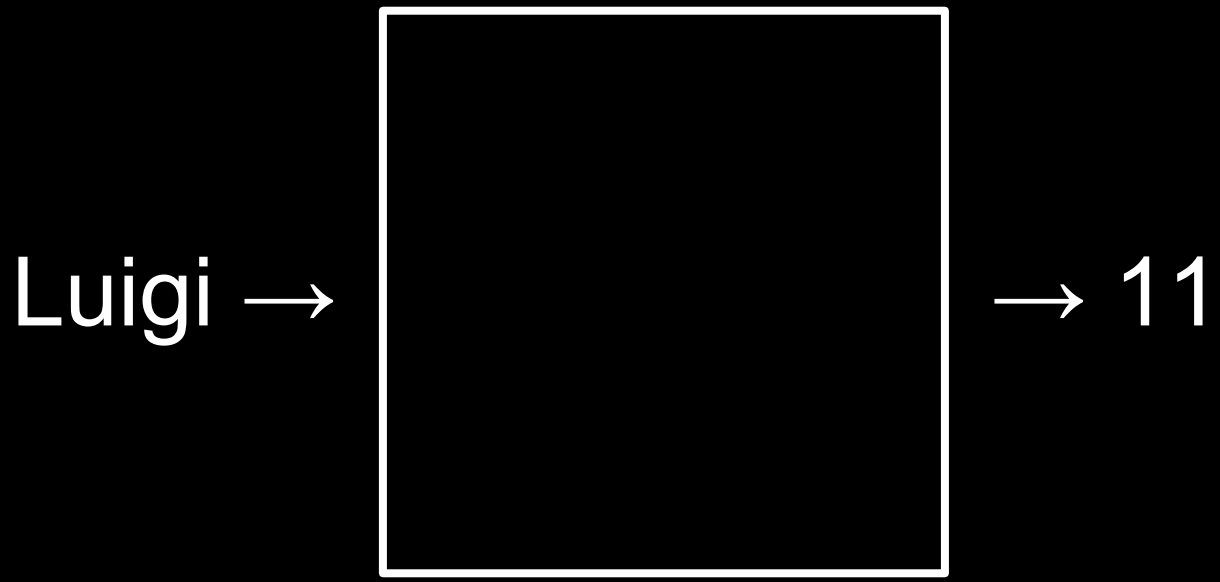
```
typedef struct node
{
    char *name;
    char *number;
    struct node *next;
} node;
```

```
node *table[26];
```


input →
→ output


Hash-Funktion


Mario →
→ 12


Luigi →
→ 11

```c
#include <ctype.h>

int hash(char *word)
{
    return toupper(word[0]) - 'A';
}
```

```c
#include <ctype.h>

int hash(const char *word)
{
    return toupper(word[0]) - 'A';
}
```

```c
#include <ctype.h>

unsigned int hash(const char *word)
{
    return toupper(word[0]) - 'A';
}
```

$O(n)$

$O(n / k)$

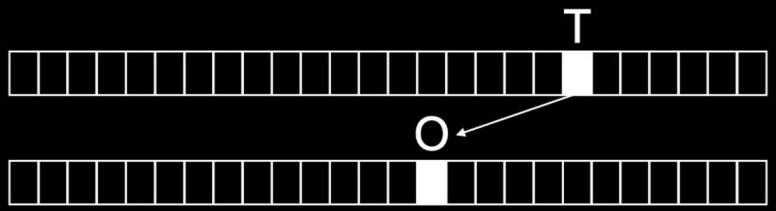
$O(n)$

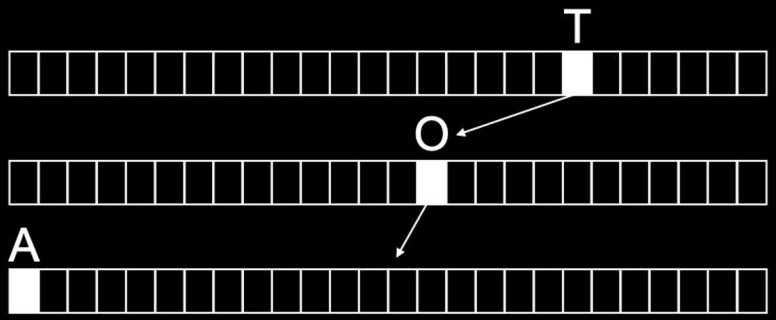
$O(1)$

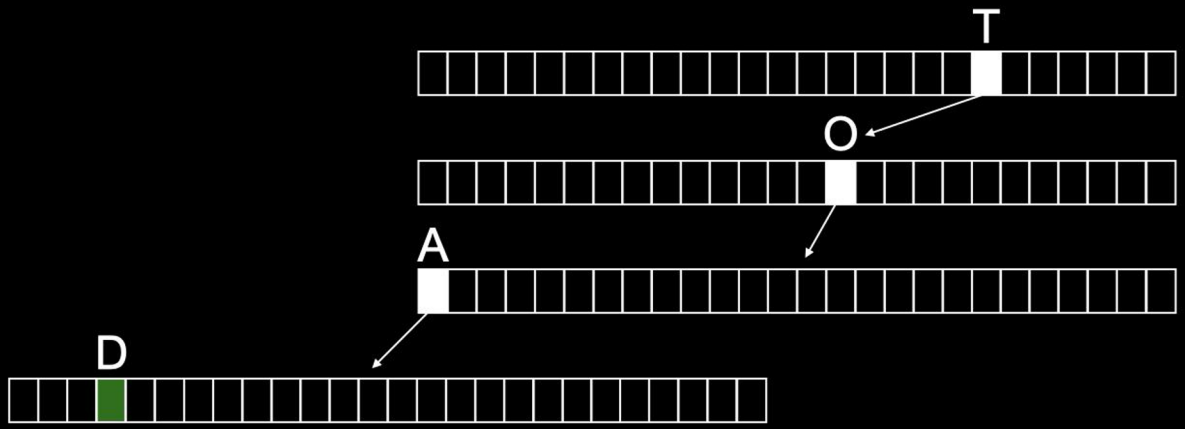
Tries

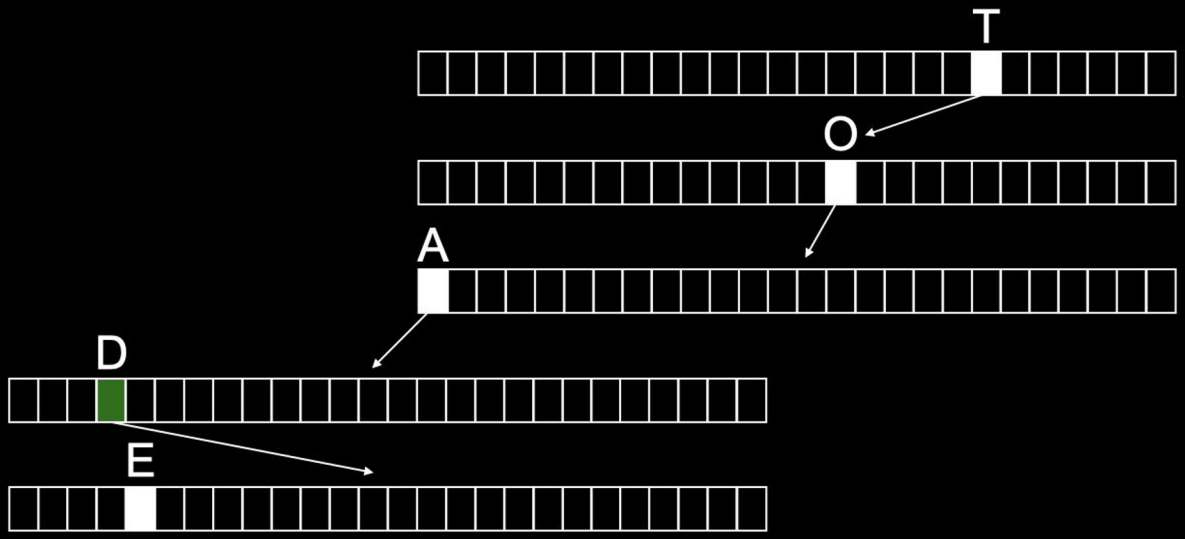

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

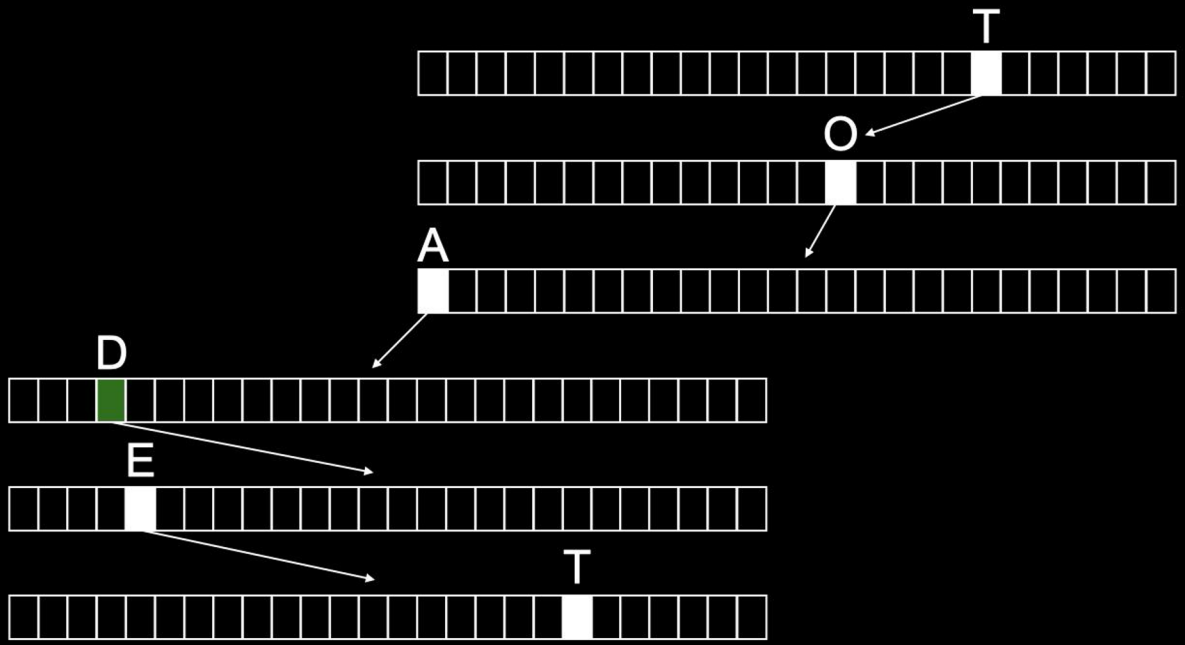

T

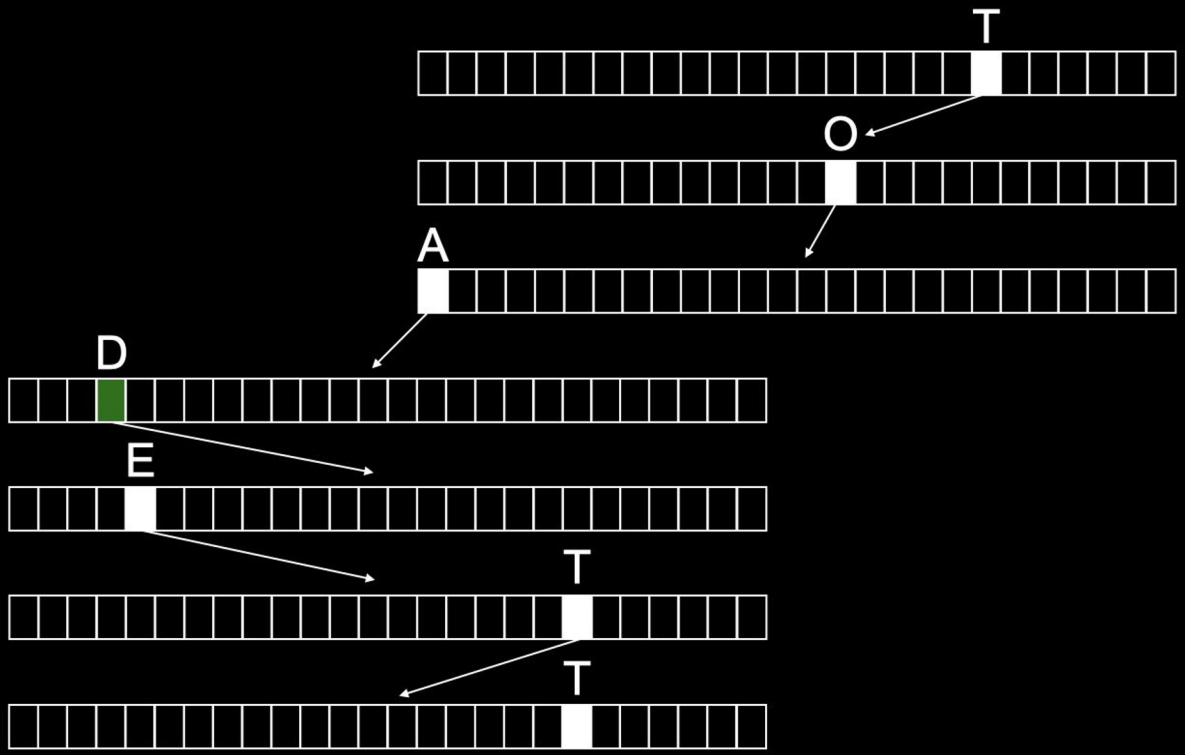

T
O

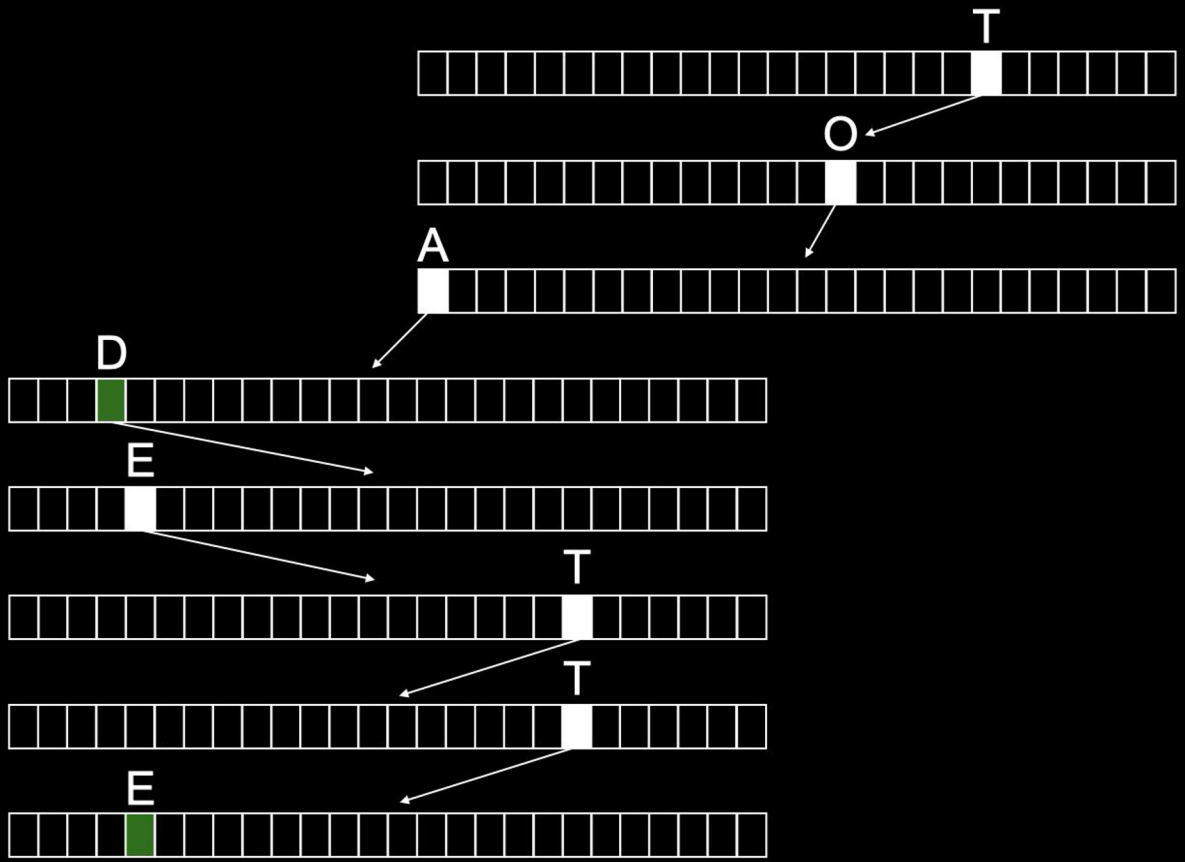

T
O
A

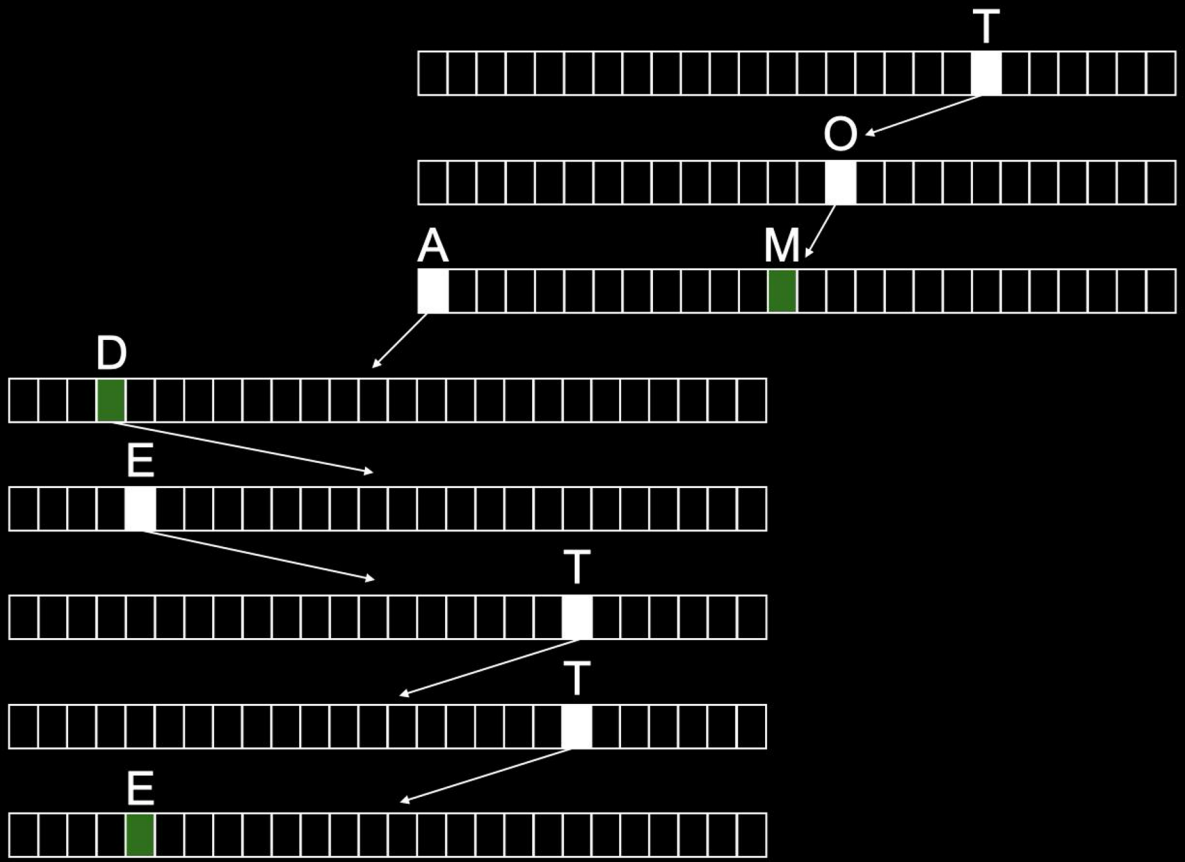

T
O
A
D


T
O
A
D
E


T
O
A
D
E
T


T
O
A
D
E
T
T


T
O
A
D
E
T
T
E


T
O
A
M
D
E
T
T
E

```
typedef struct node
{
    struct node *children[26];
    char *number;
} node;
```

```
node *trie;
```

$O(n^2)$

$O(n \log n)$

$O(n)$

$O(\log n)$

$O(1)$

Hoher Platzbedarf, aber konstante Laufzeit.

Effizientere Varianten im Short zu Tries.

Dies war Inf-Einf-B.