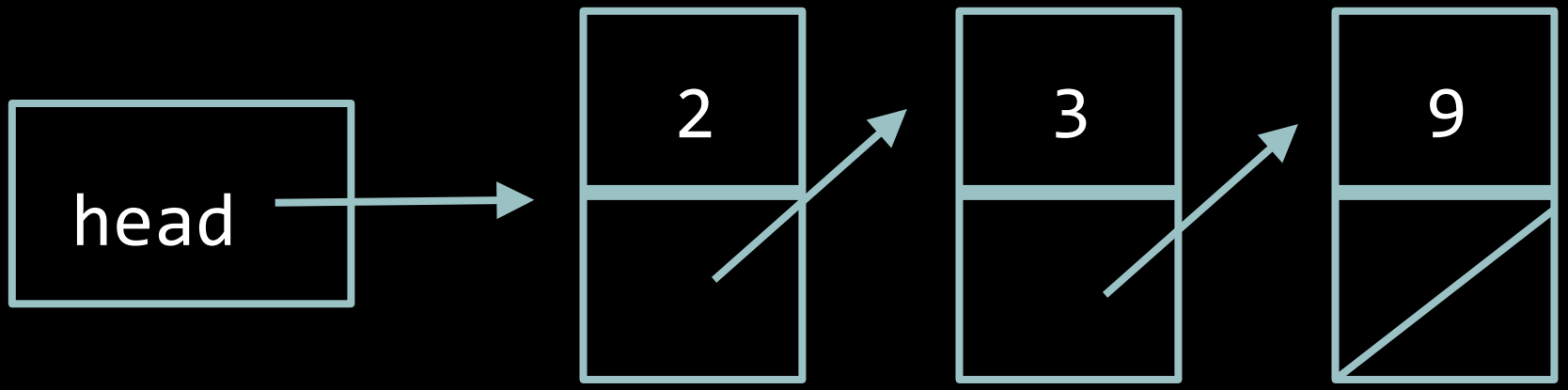
Agenda

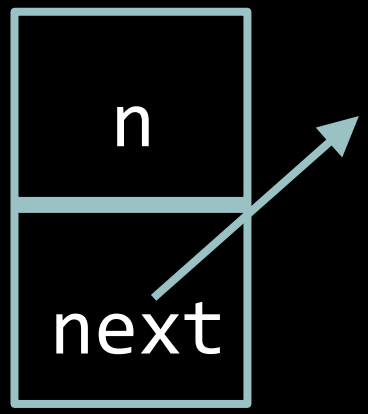
- Linked lists
- Hash Tables
- Tries
- Binary Trees
- Stacks
- Queues

- Quizzes!

Linked Lists

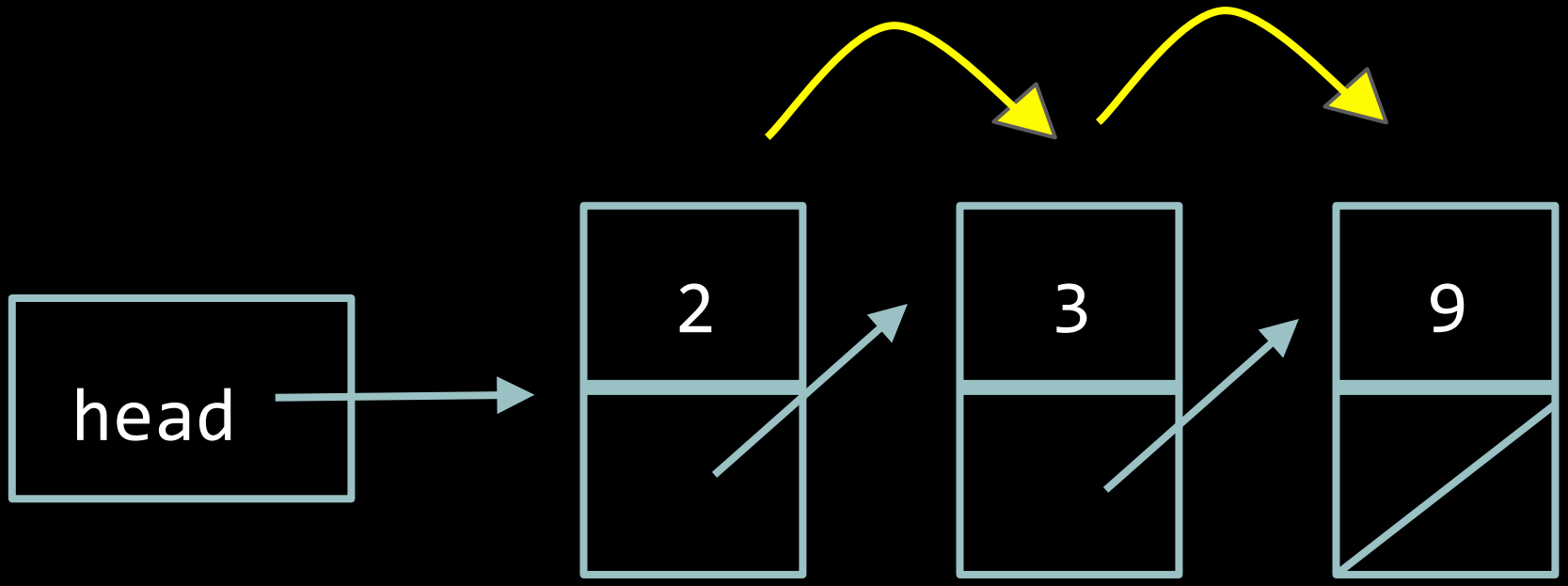


Nodes



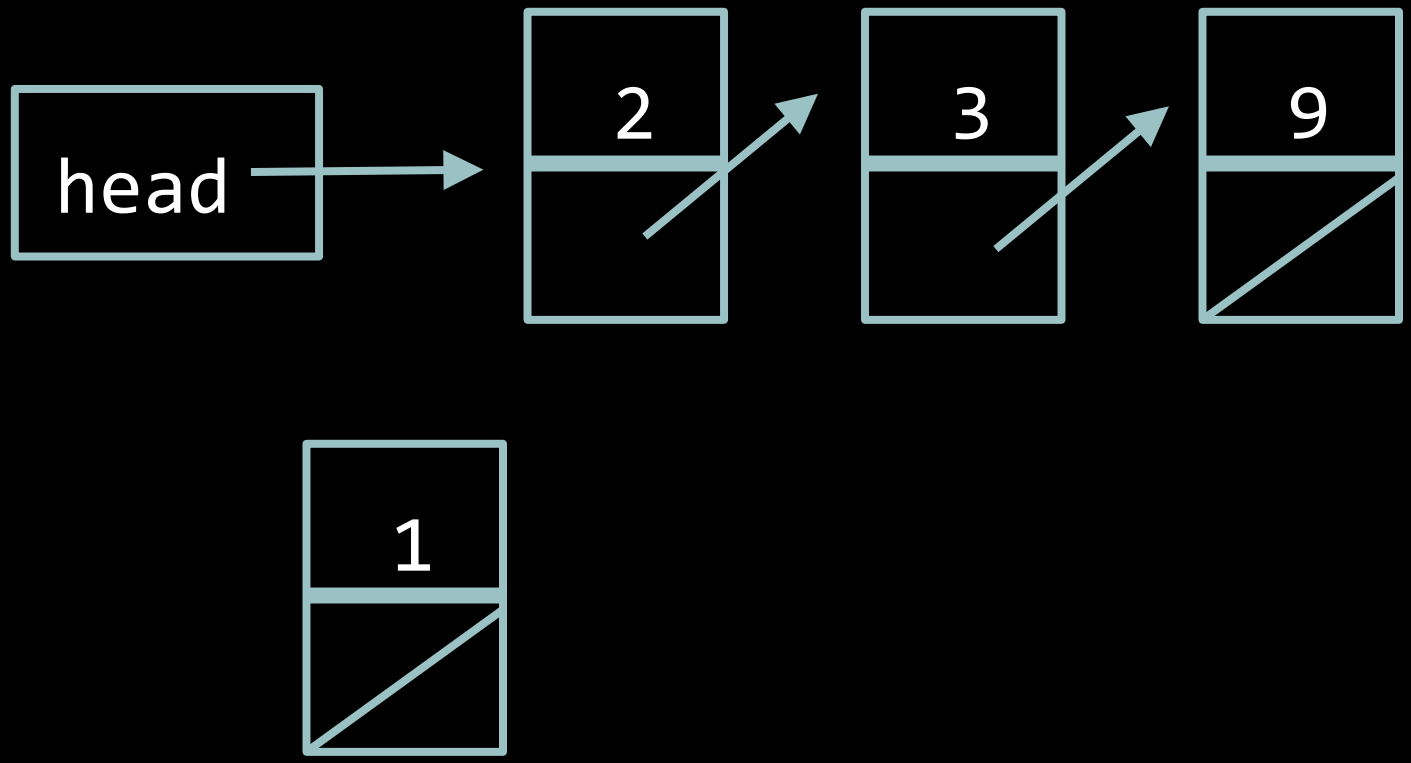

```
typedef struct node
{
        int n;
        struct node*
next;
}
node;
```

Search

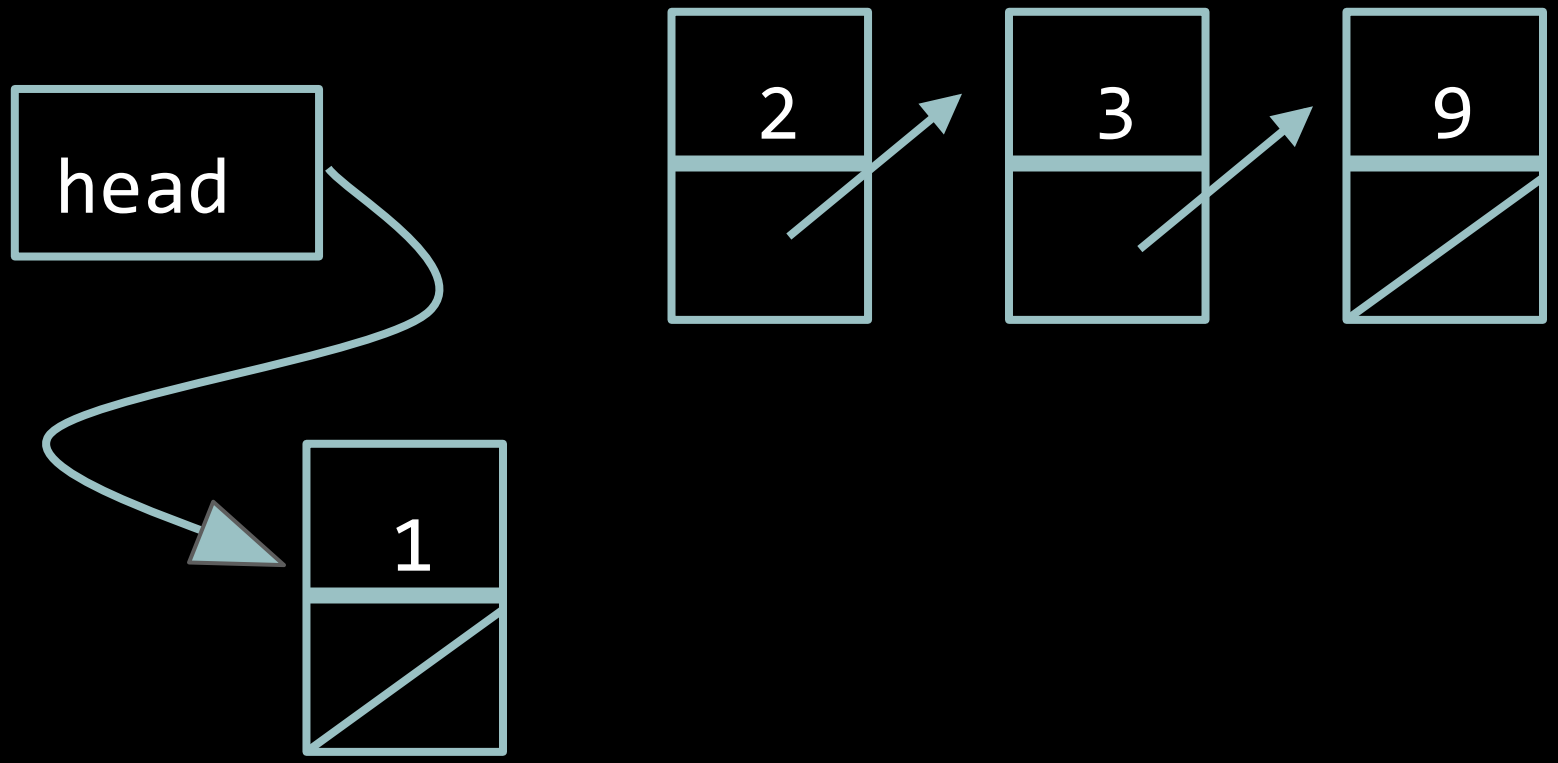


```
bool search(node* list, int n)
{
    node* ptr = list;

    while (ptr != NULL)
    {
            if (ptr->n == n)
      {
                return true;
      }
            ptr = ptr->next;
    }
    return false;
}
```

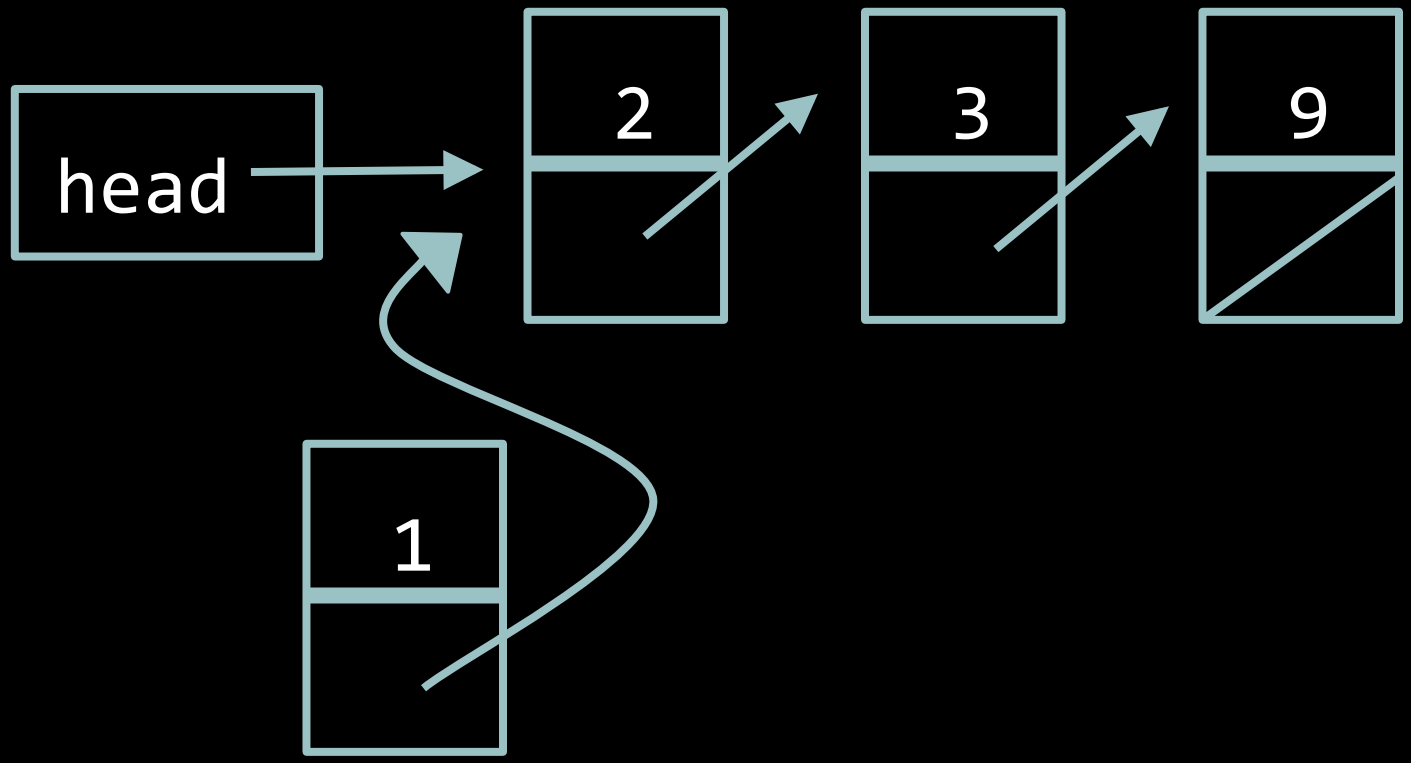
Insertion



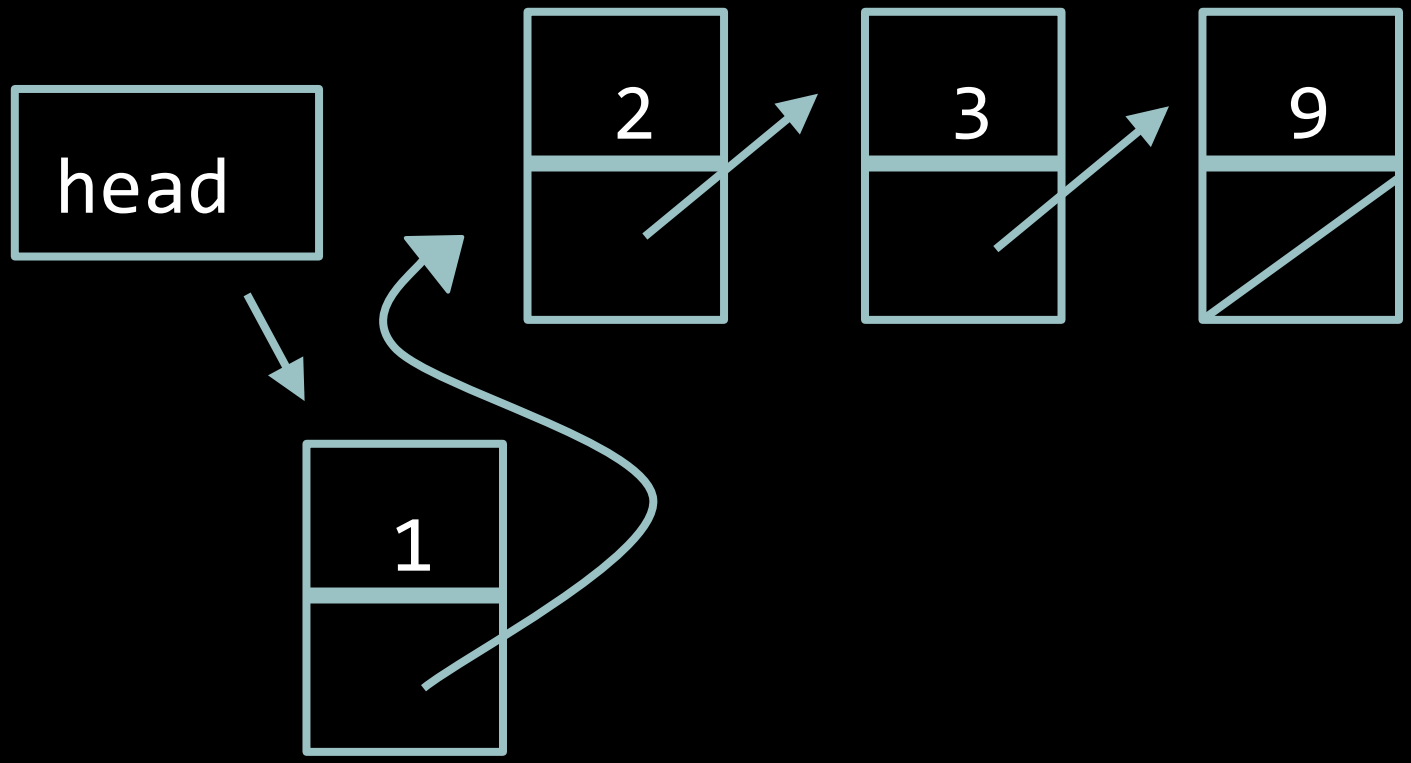
Insertion



Insertion

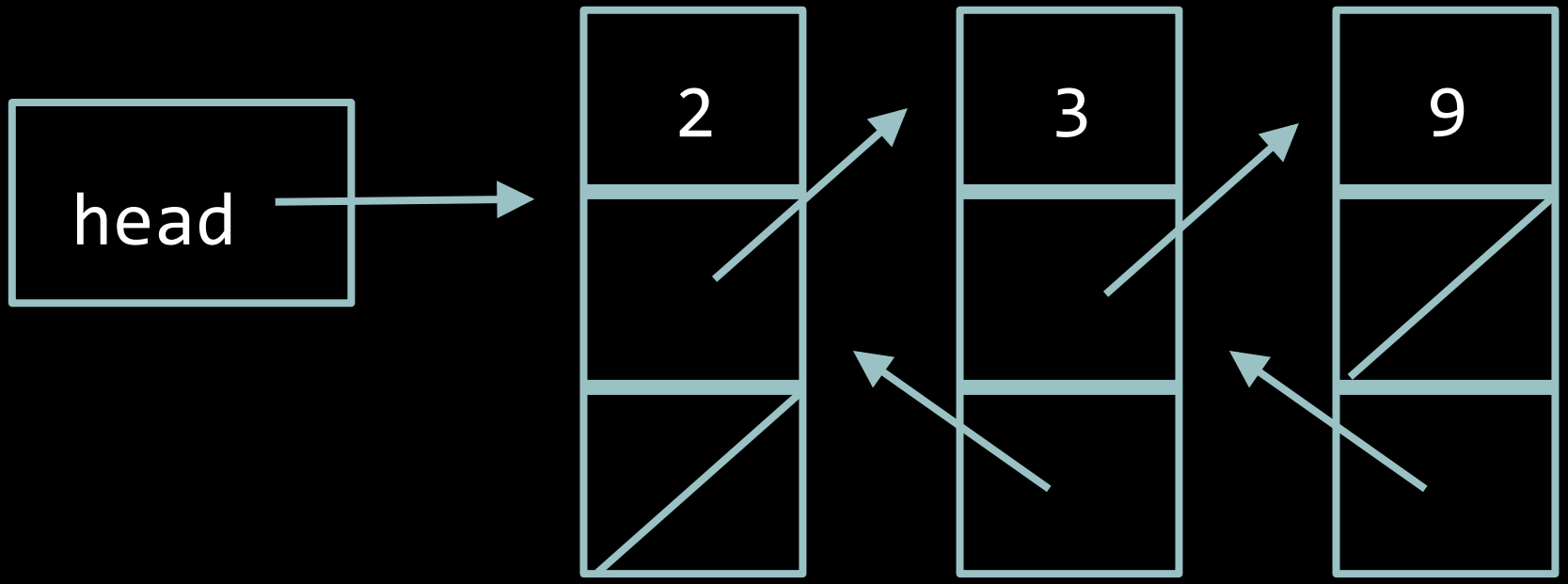


Insertion

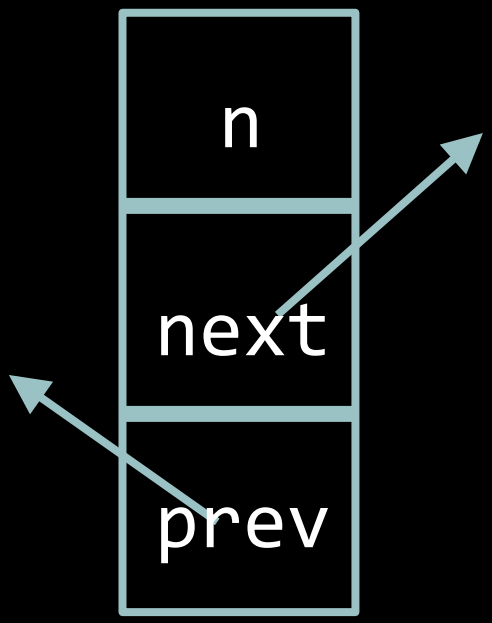


```c
void insert(int n)
{
    // create new node
    node* new = malloc(sizeof(node));

    // check for NULL
    if (new == NULL)
    {
        exit(1);
    }
    // initialize new node
    new->n = n;
    new->next = NULL;

    // insert new node at head
    new->next = head;
    head = new;
}
```

Doubly Linked Lists

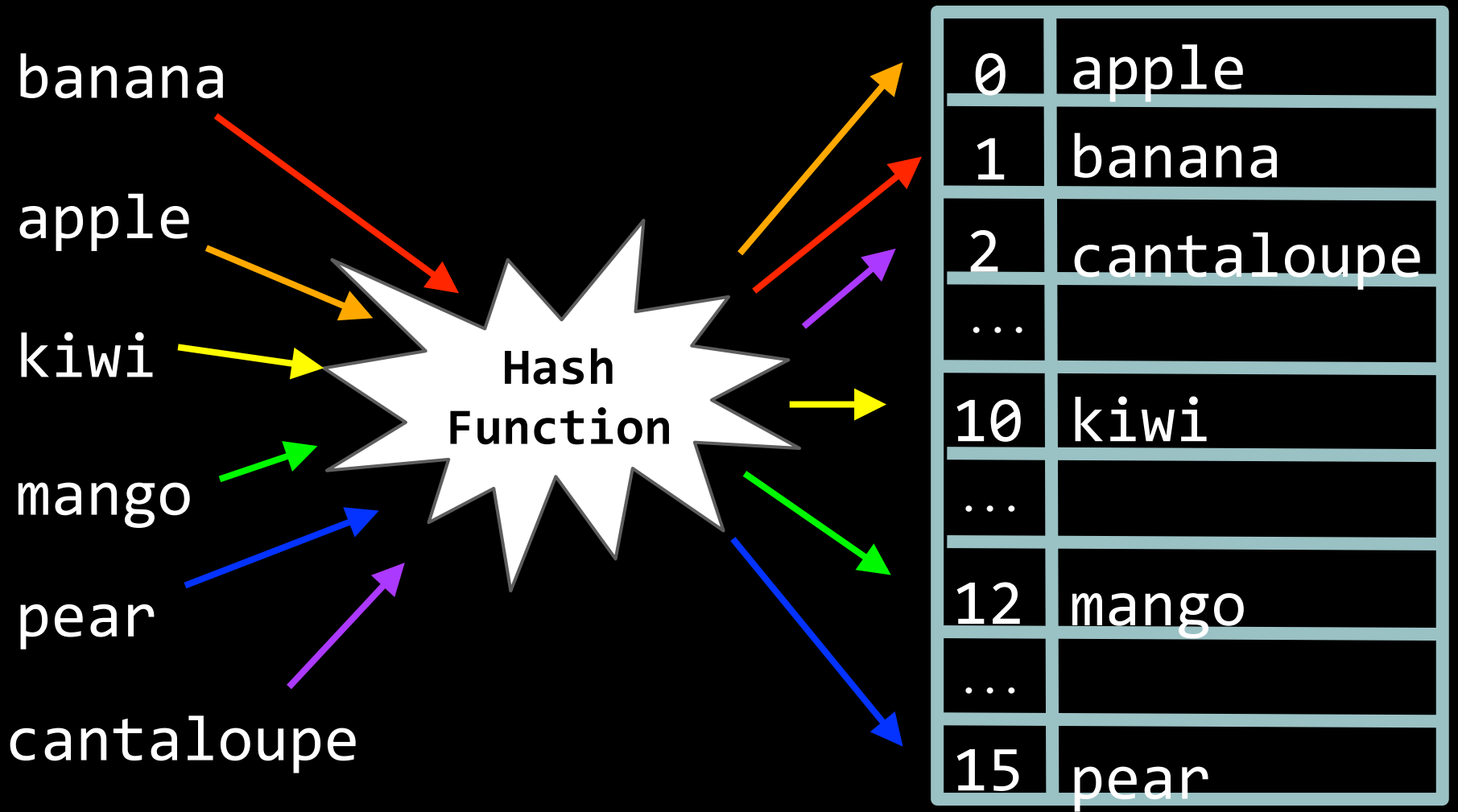


DLL Nodes

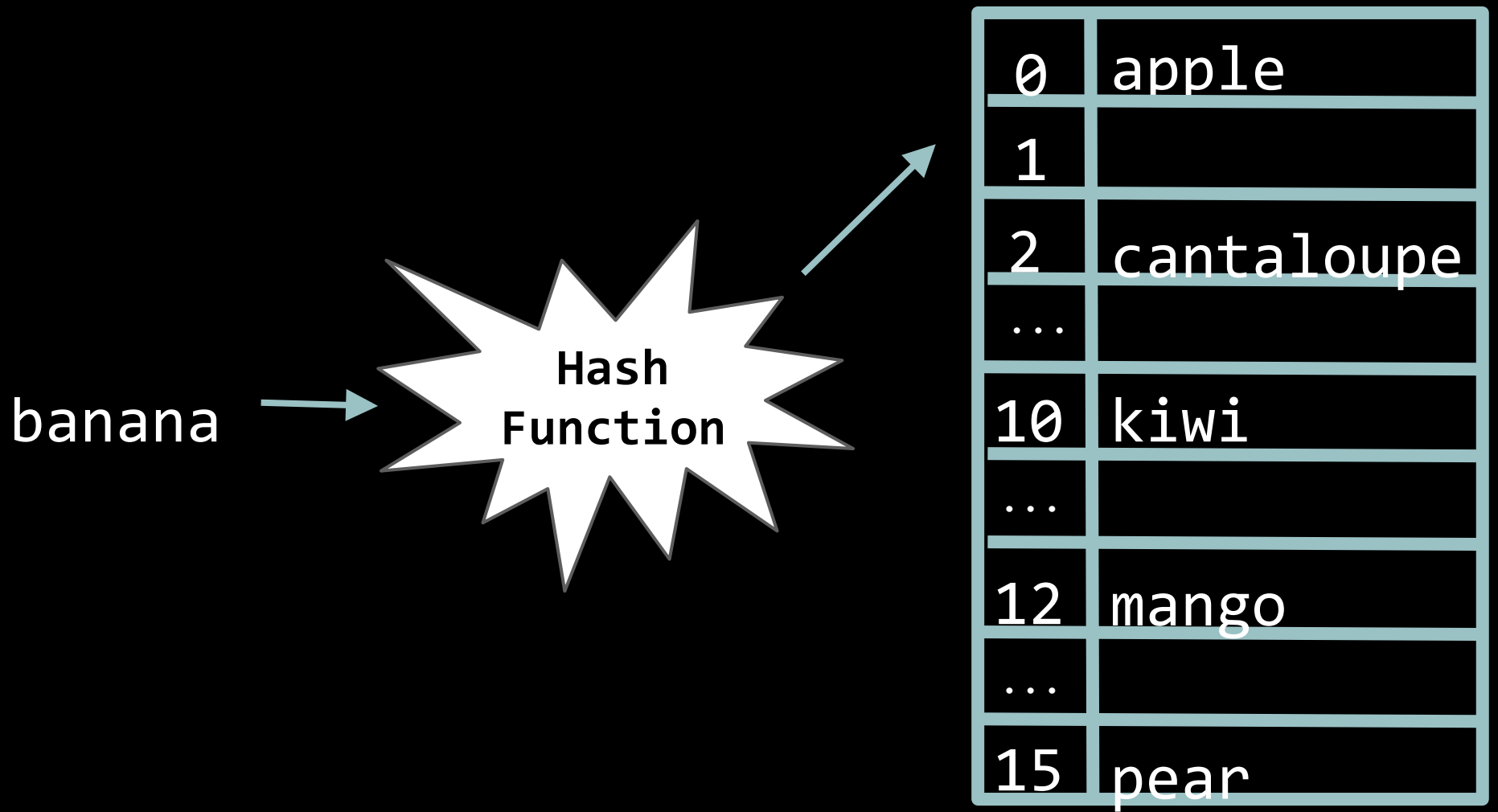



```
typedef struct  node
{
        int n;
        struct node* next;
        struct node*
prev;
}
node;
```

Hash Tables


banana
apple
kiwi
mango
pear
cantaloupe
Hash Function
0 apple
1 banana
2 cantaloupe
...
10 kiwi
...
12 mango
...
15 pear

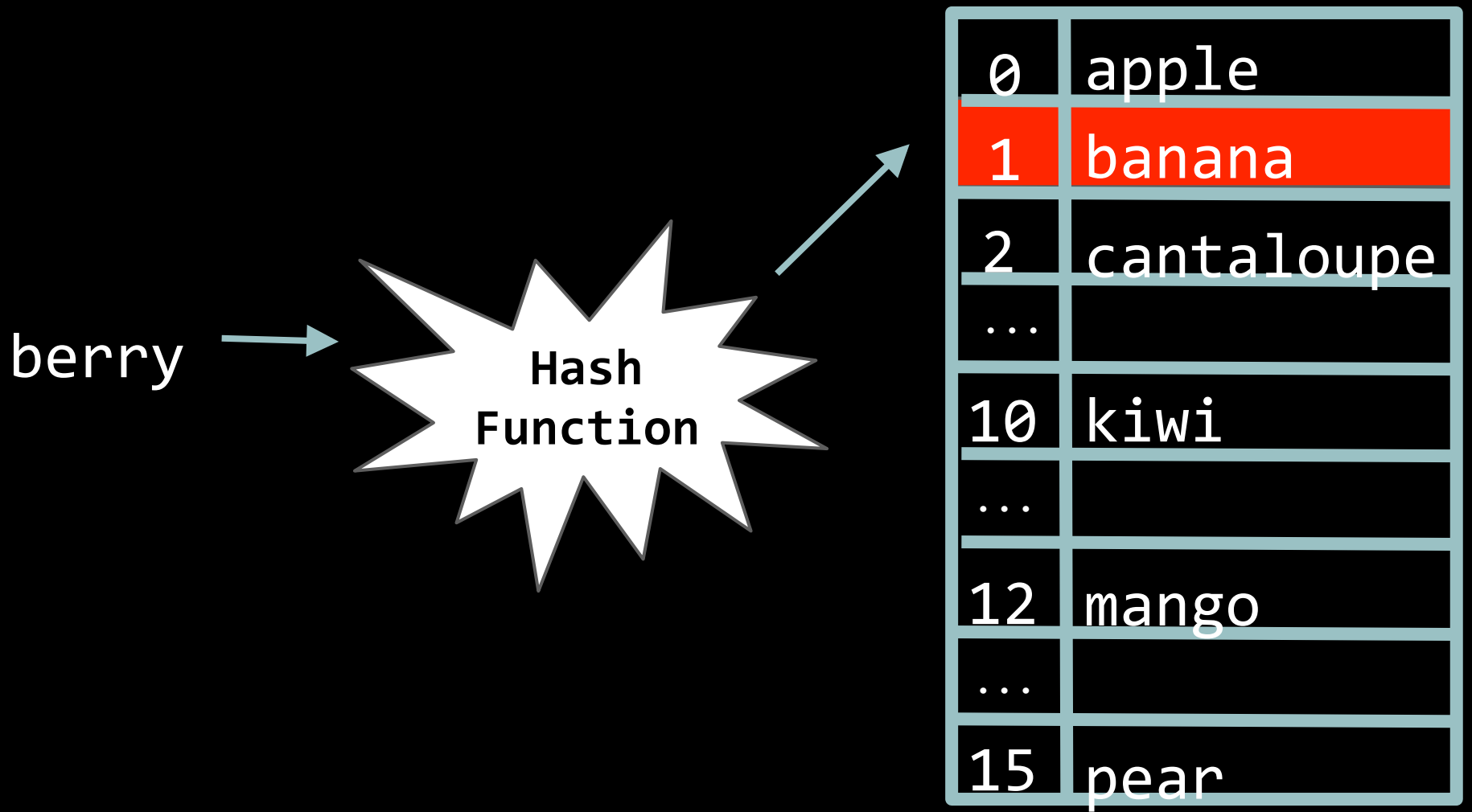
Hash Function



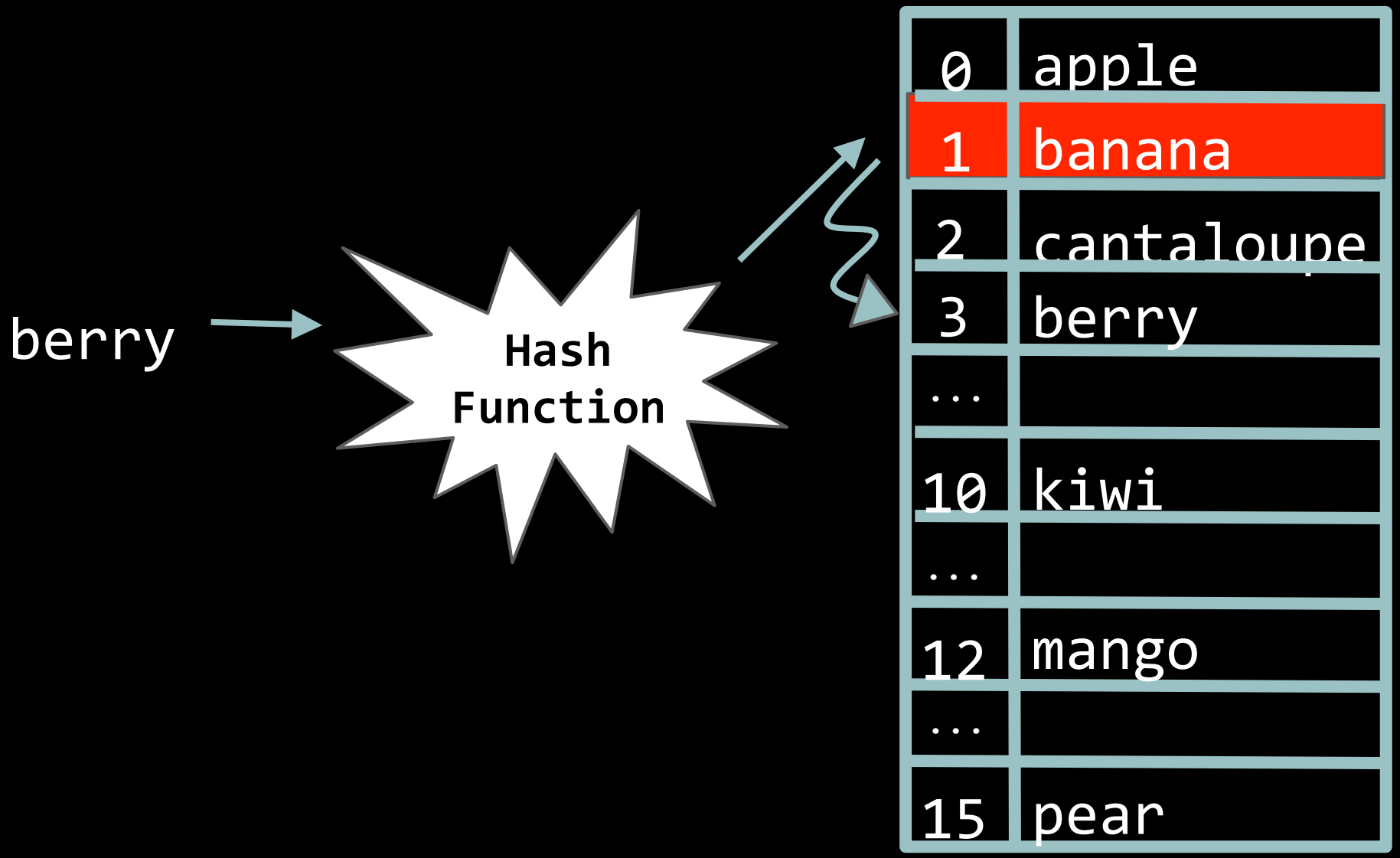
Hash Function Example

```
int hash_function(char* key)
{
        // hash on first letter of string
        int hash = toupper(key[0]) - 'A';

    return hash % SIZE;
}
```

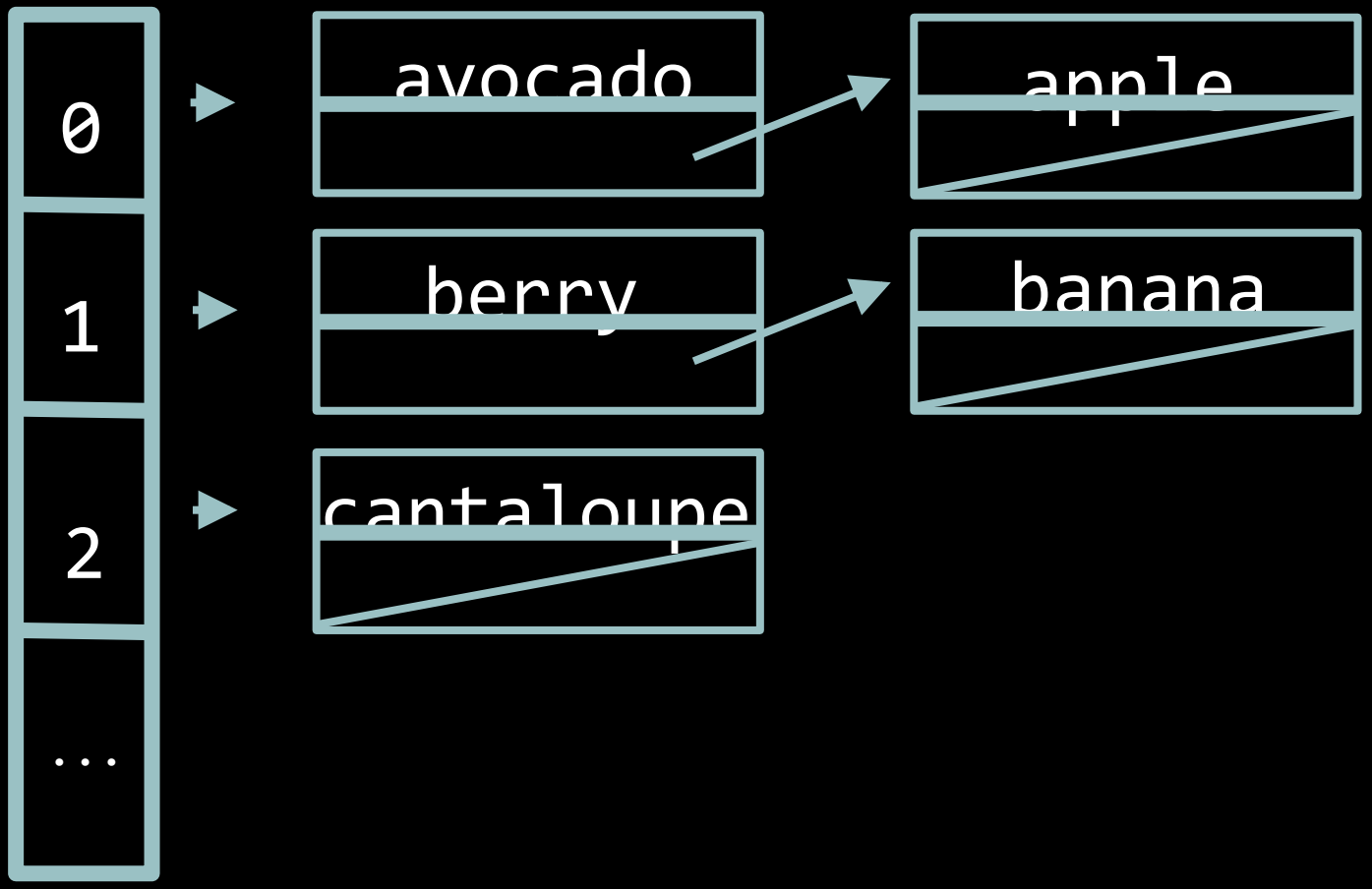
Collisions


berry
Hash Function
0 apple
1 banana
2 cantaloupe
...
10 kiwi
...
12 mango
...
15 pear

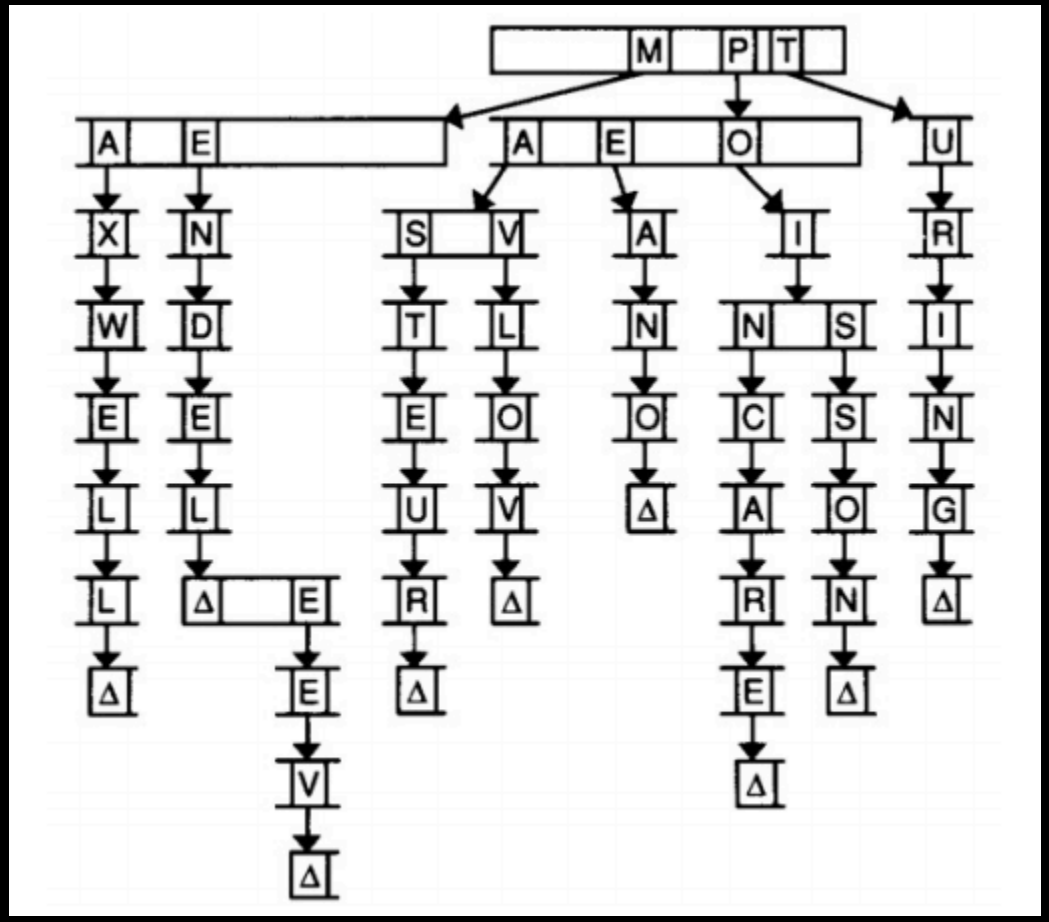
Linear Probing



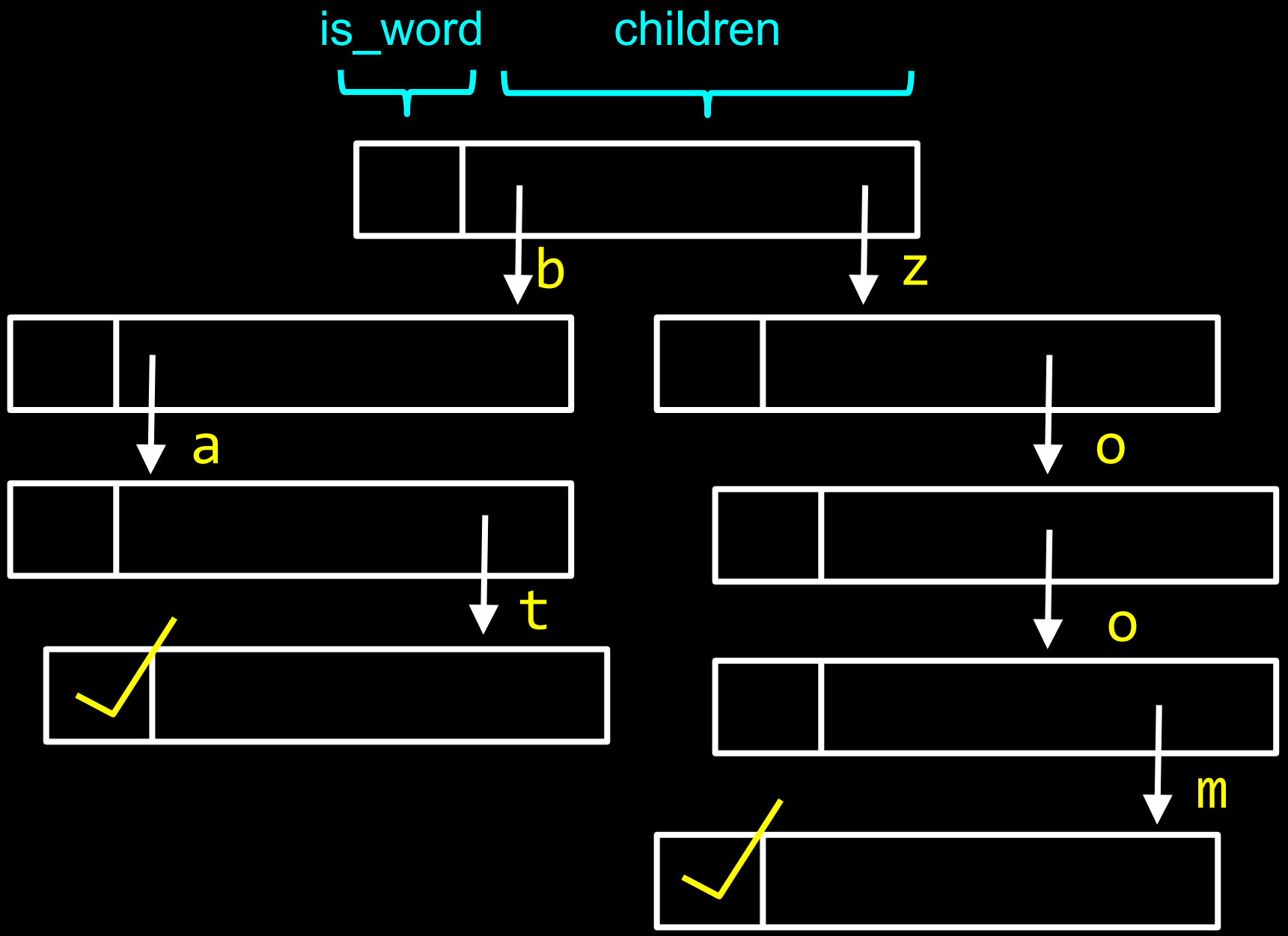
Separate Chaining

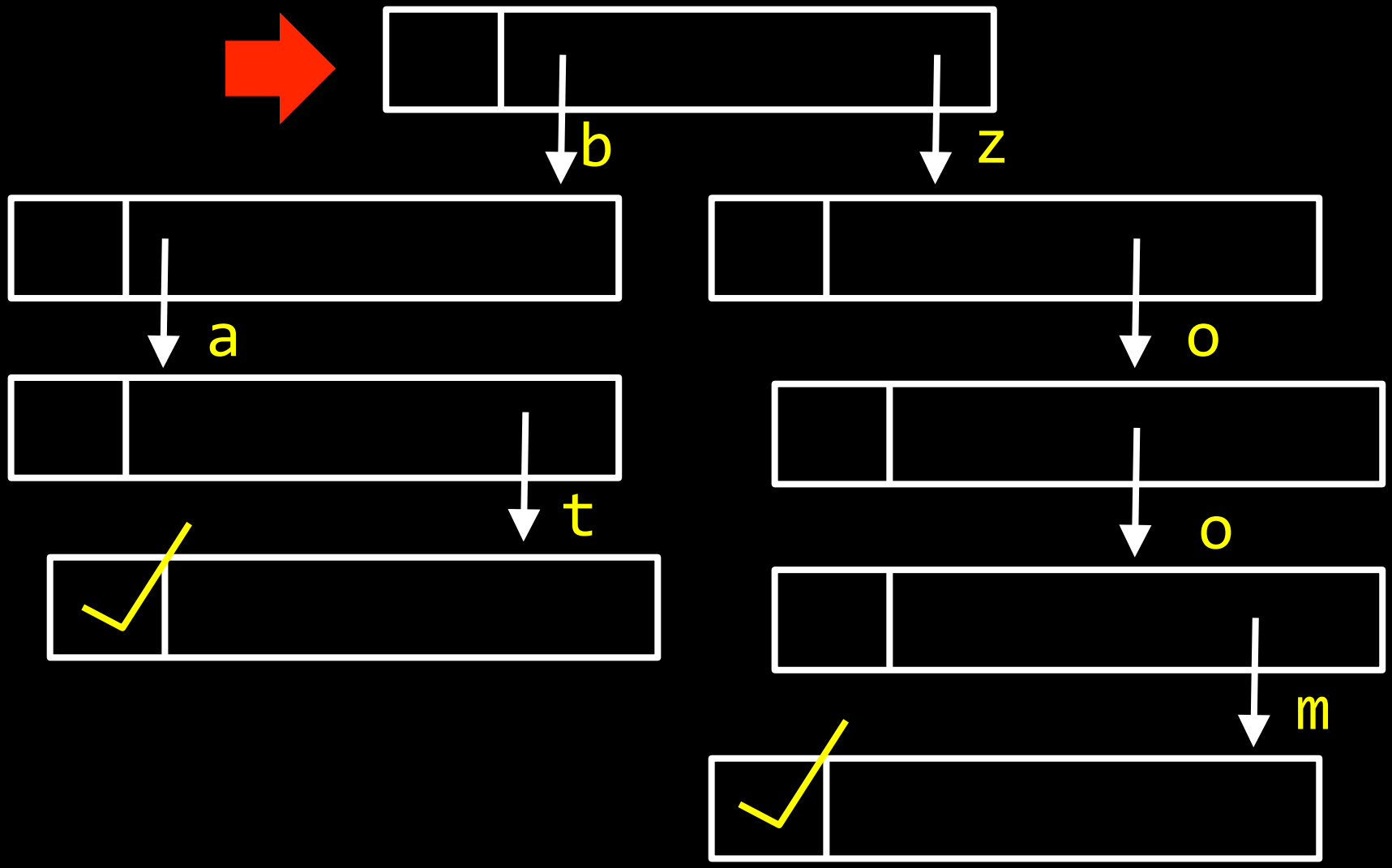


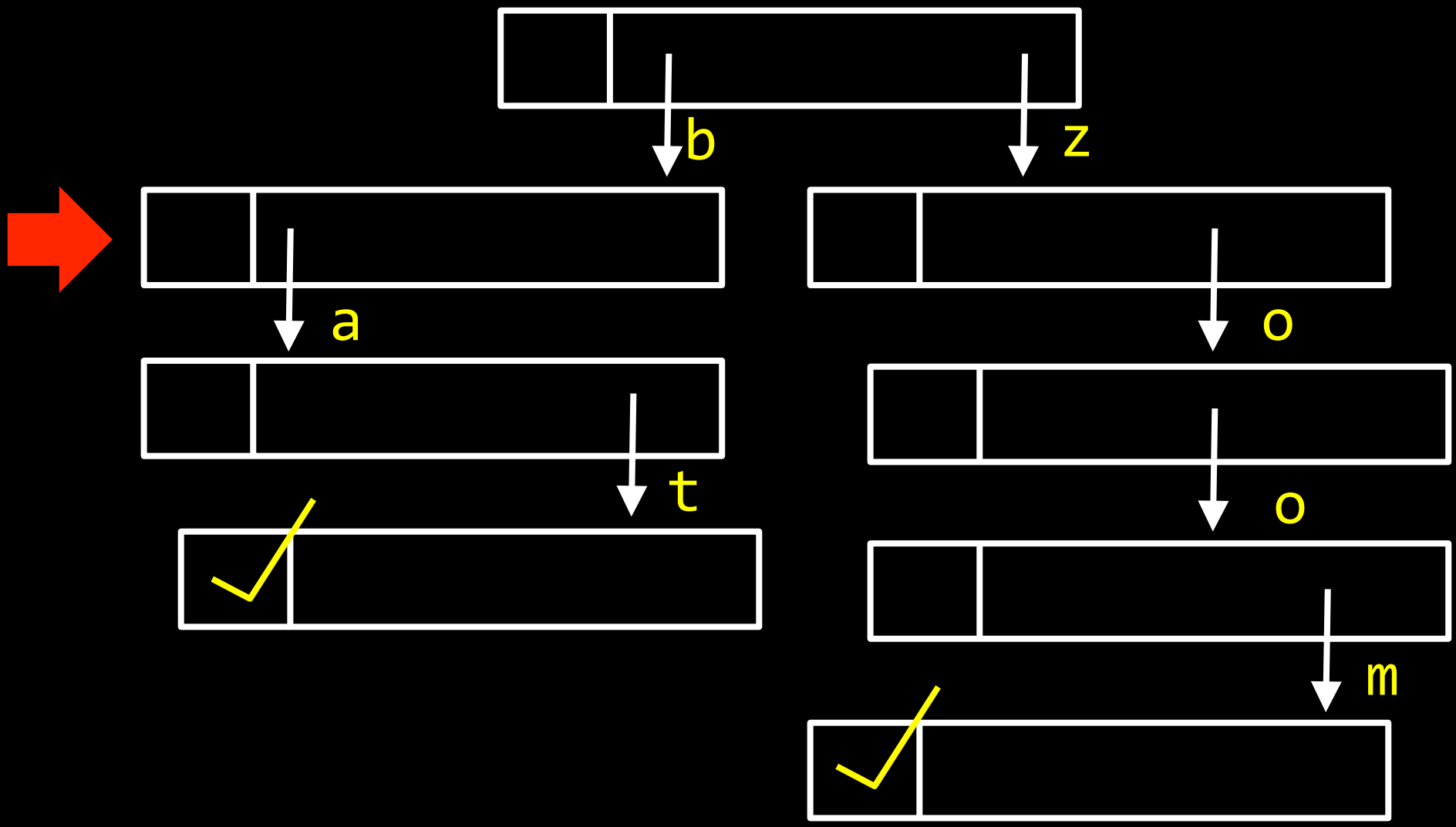
Tries

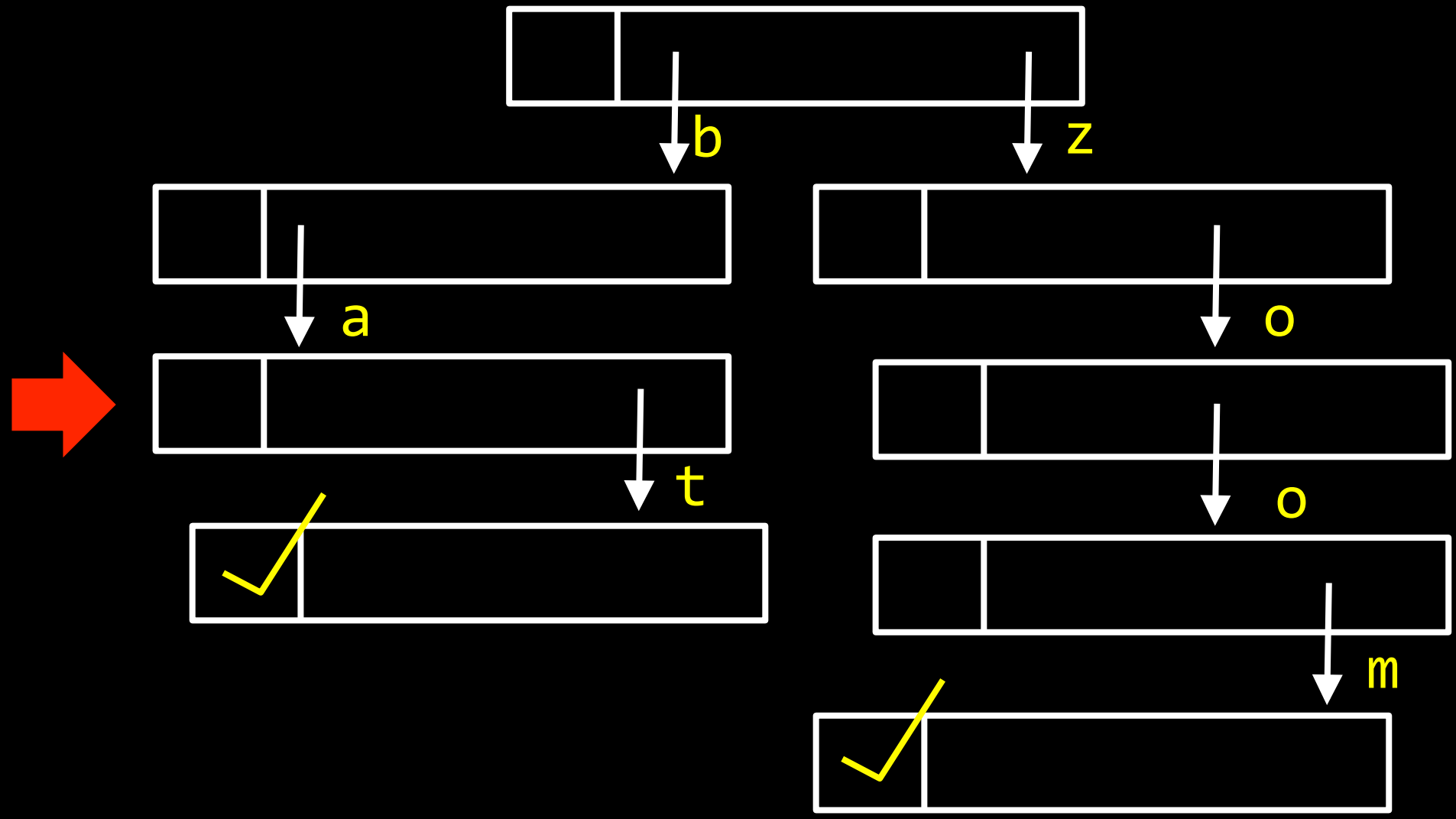

M P T
A E
A E O
U
X N
S V
A
I
R
W D
T L
N
N S
I
E E
E O
O
C S
N
L L
U V
Δ
A O
G
L Δ E
R
Δ
R N
Δ
Δ
E
Δ
E
Δ
V
Δ

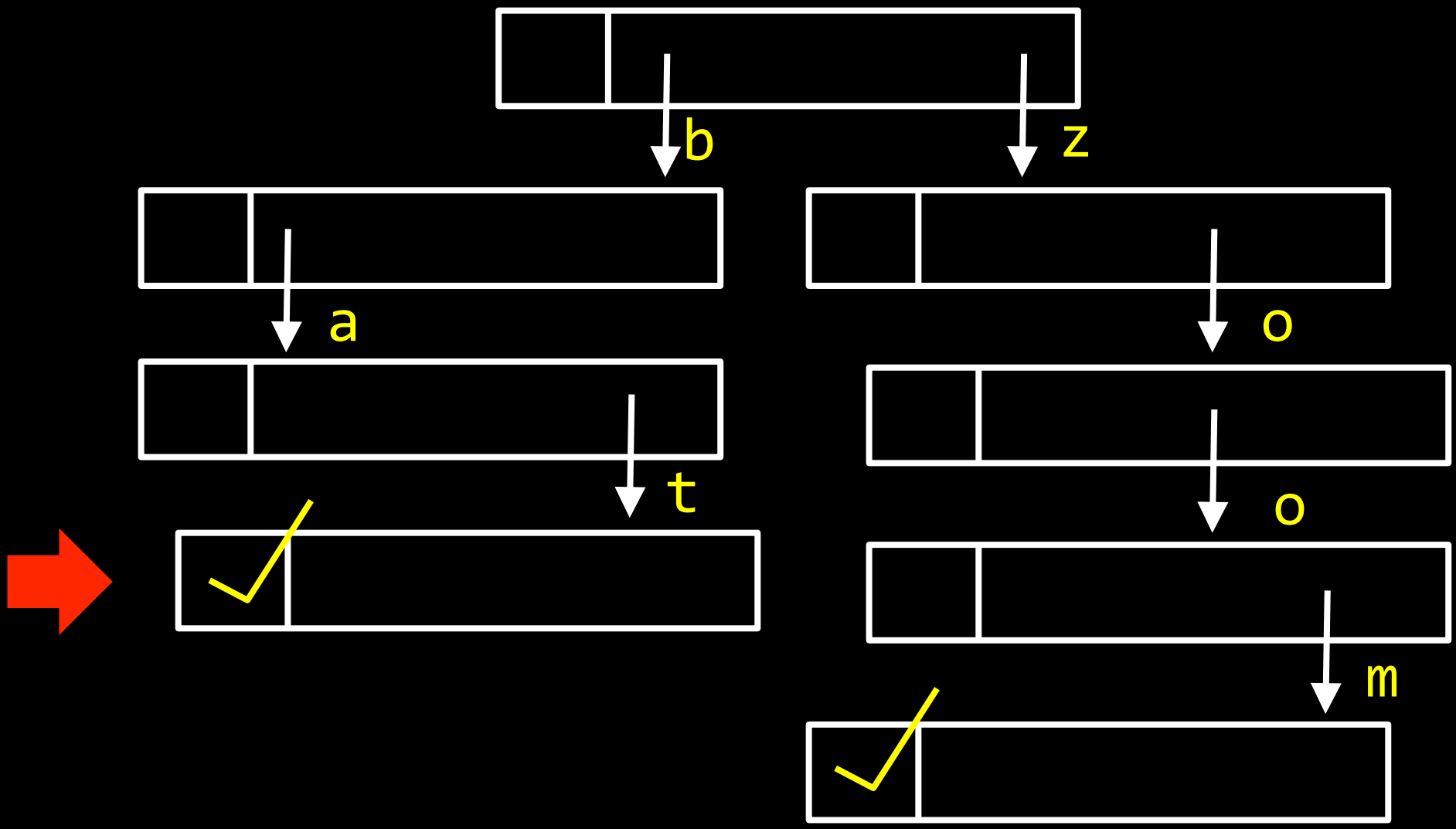
```
typedef struct node
{
    // marker for end of word
     bool is_word;

     // pointers to other nodes
     struct node* children[27];
}
node;
```

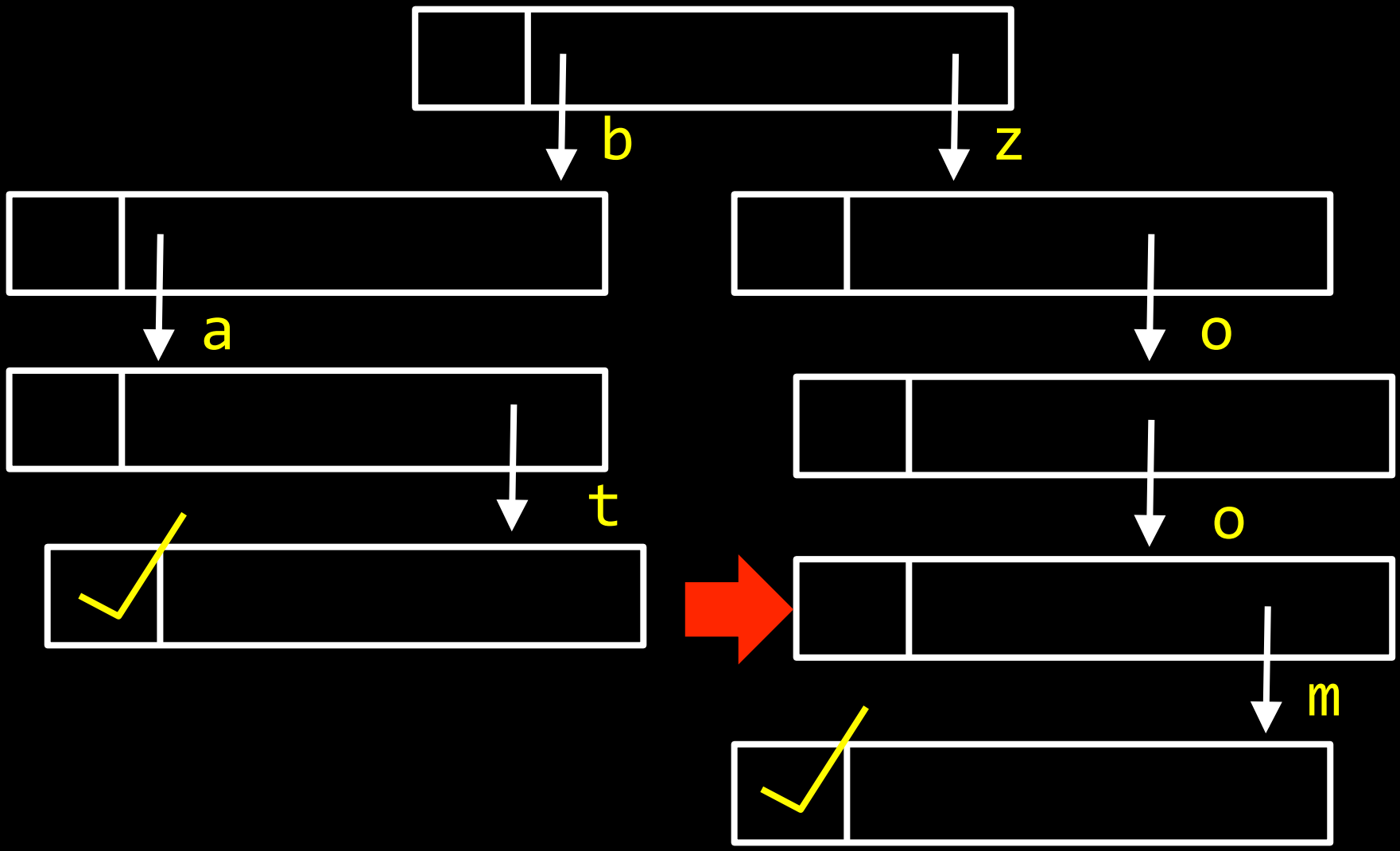

is_word
children
b
z
a
o
t
o
m

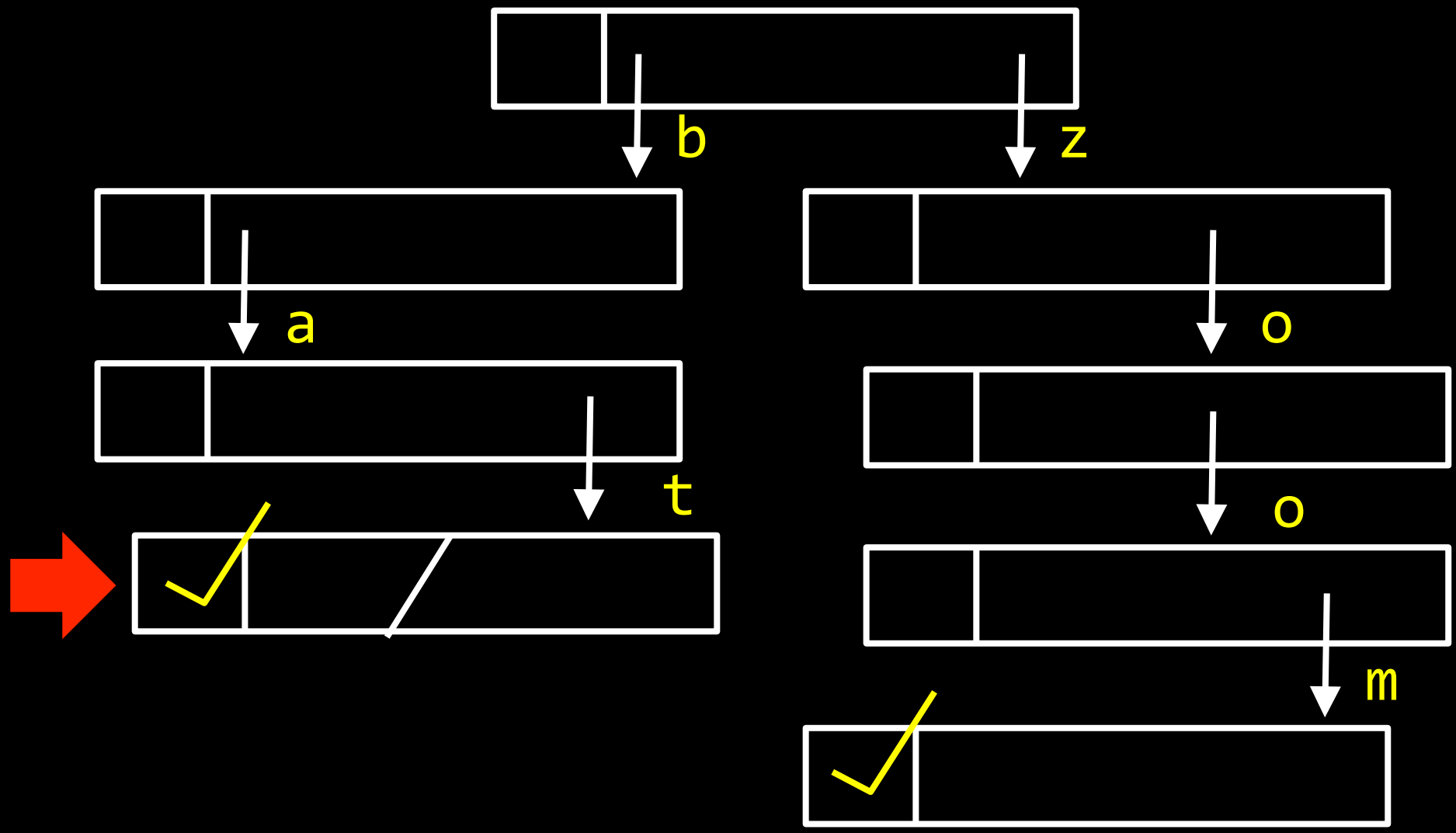

b
z
a
o
t
o
m

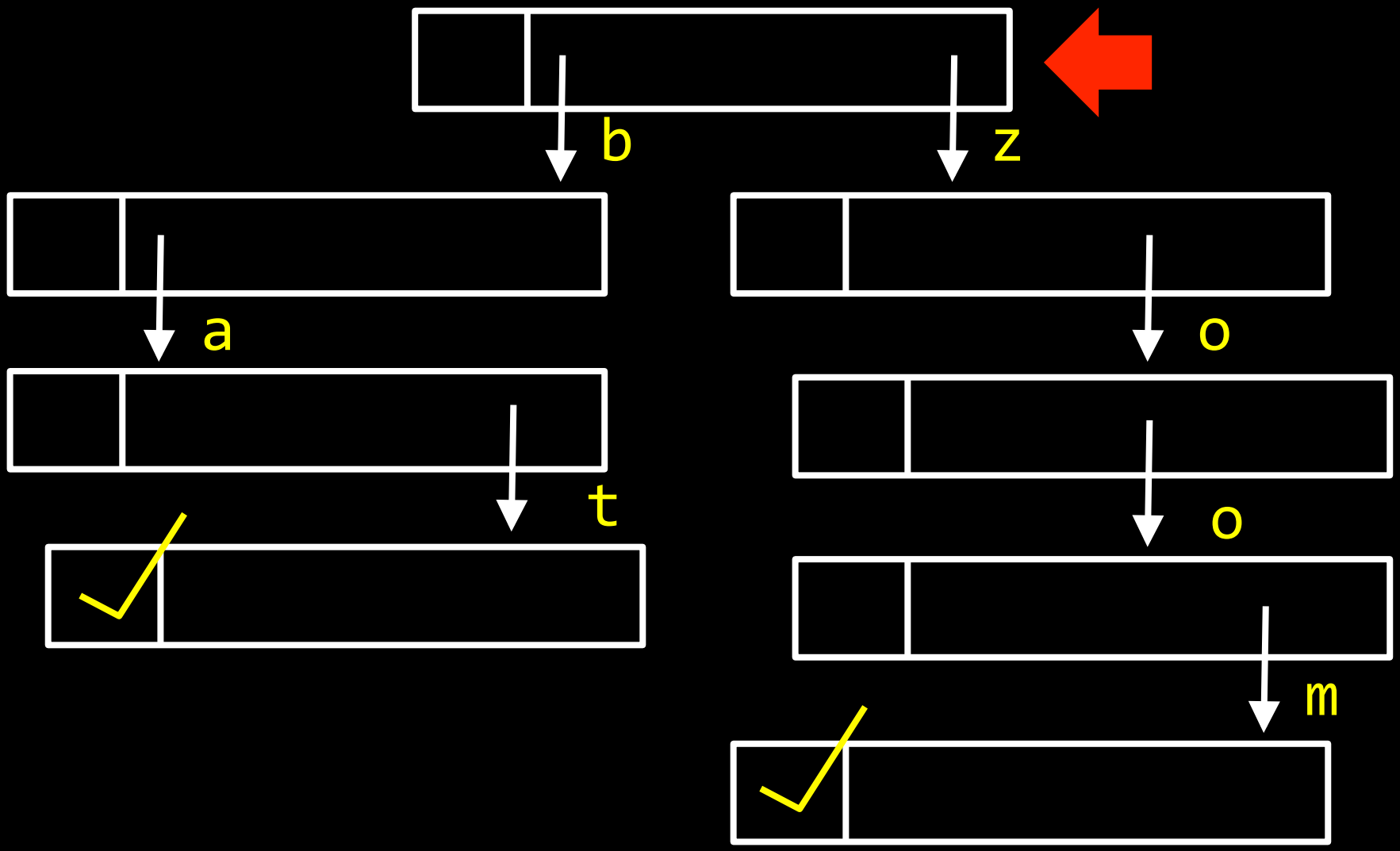

b
z
a
o
t
o
m

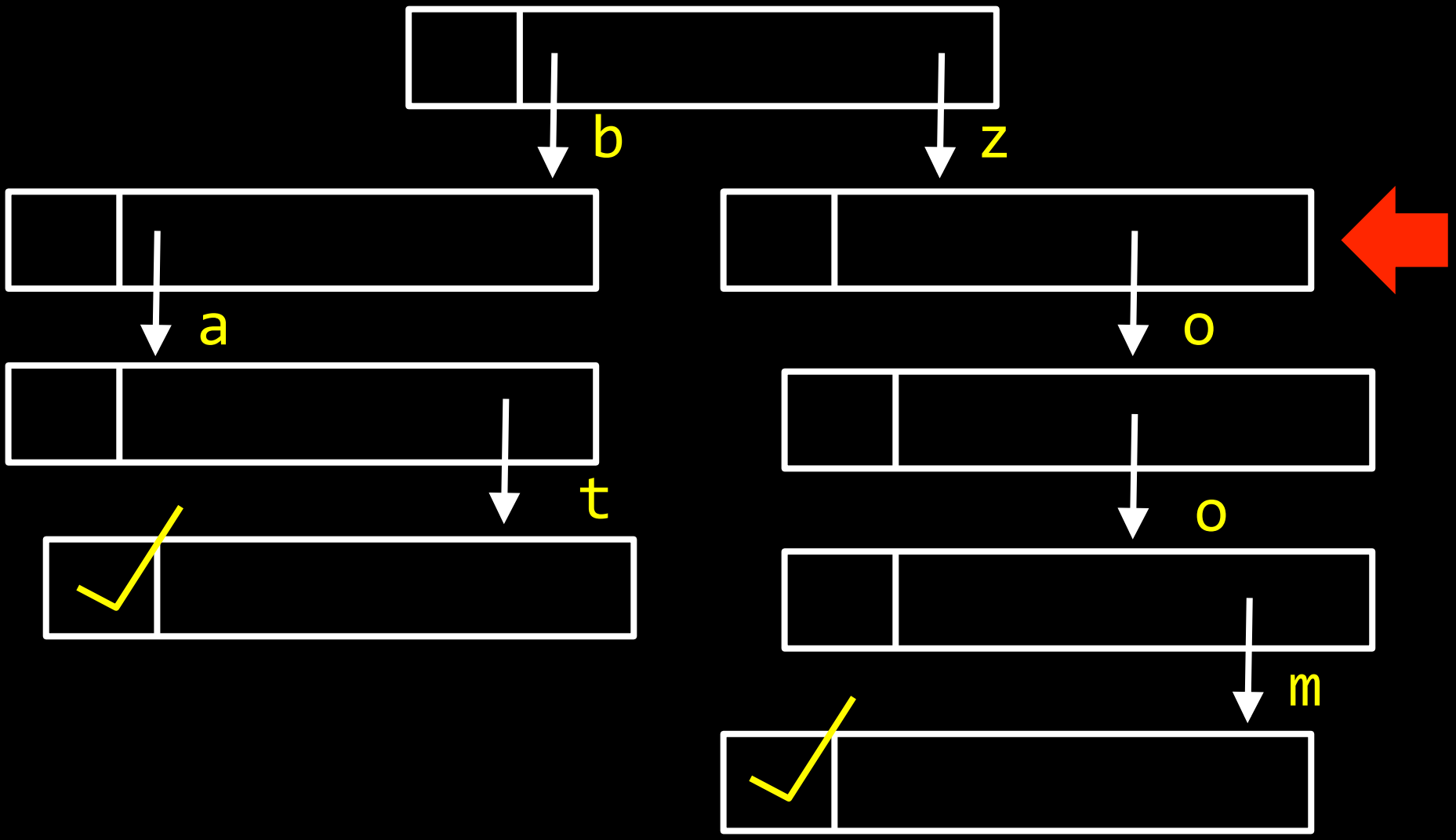

b
z
a
o
t
o
m

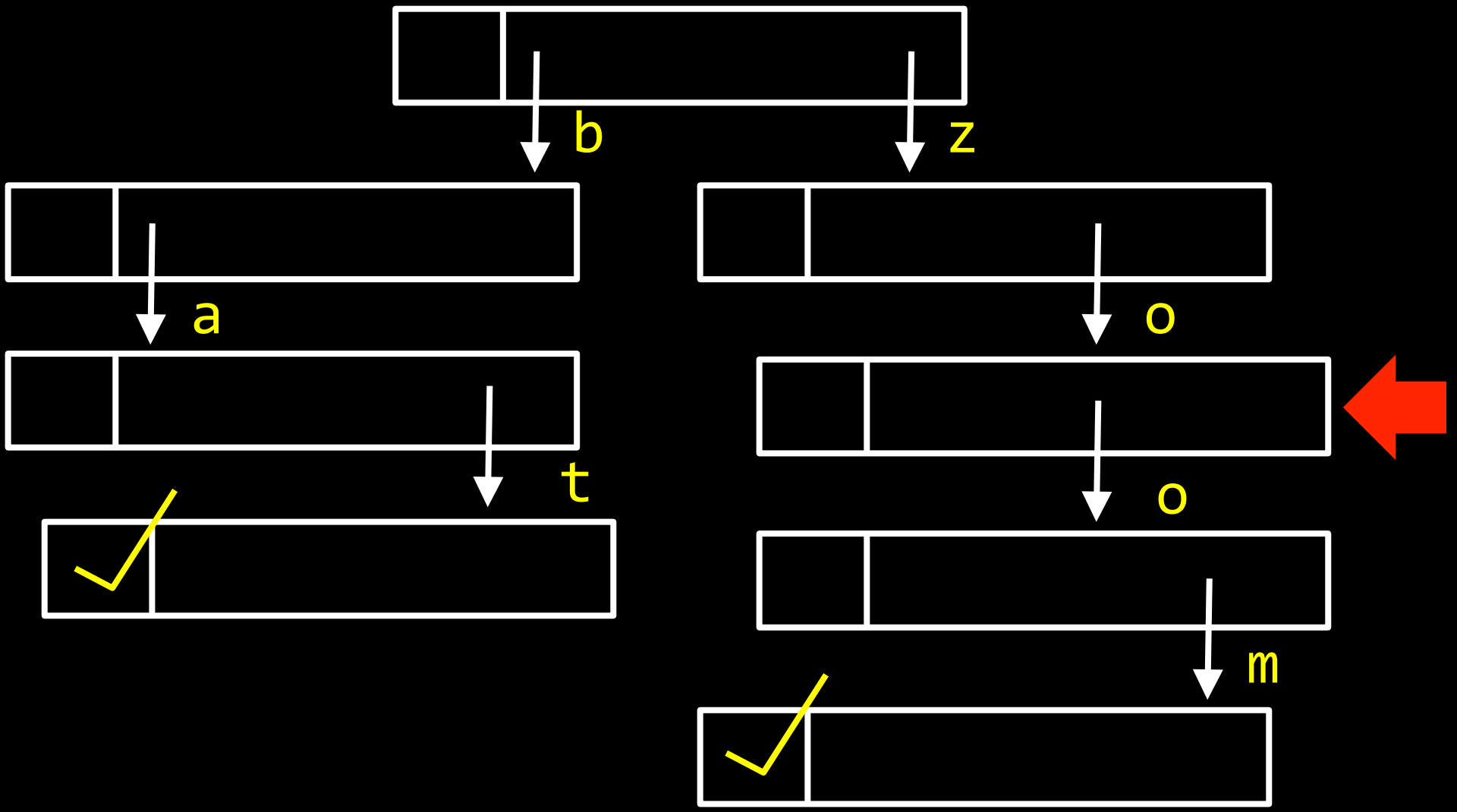

b
z
a
o
t
o
m

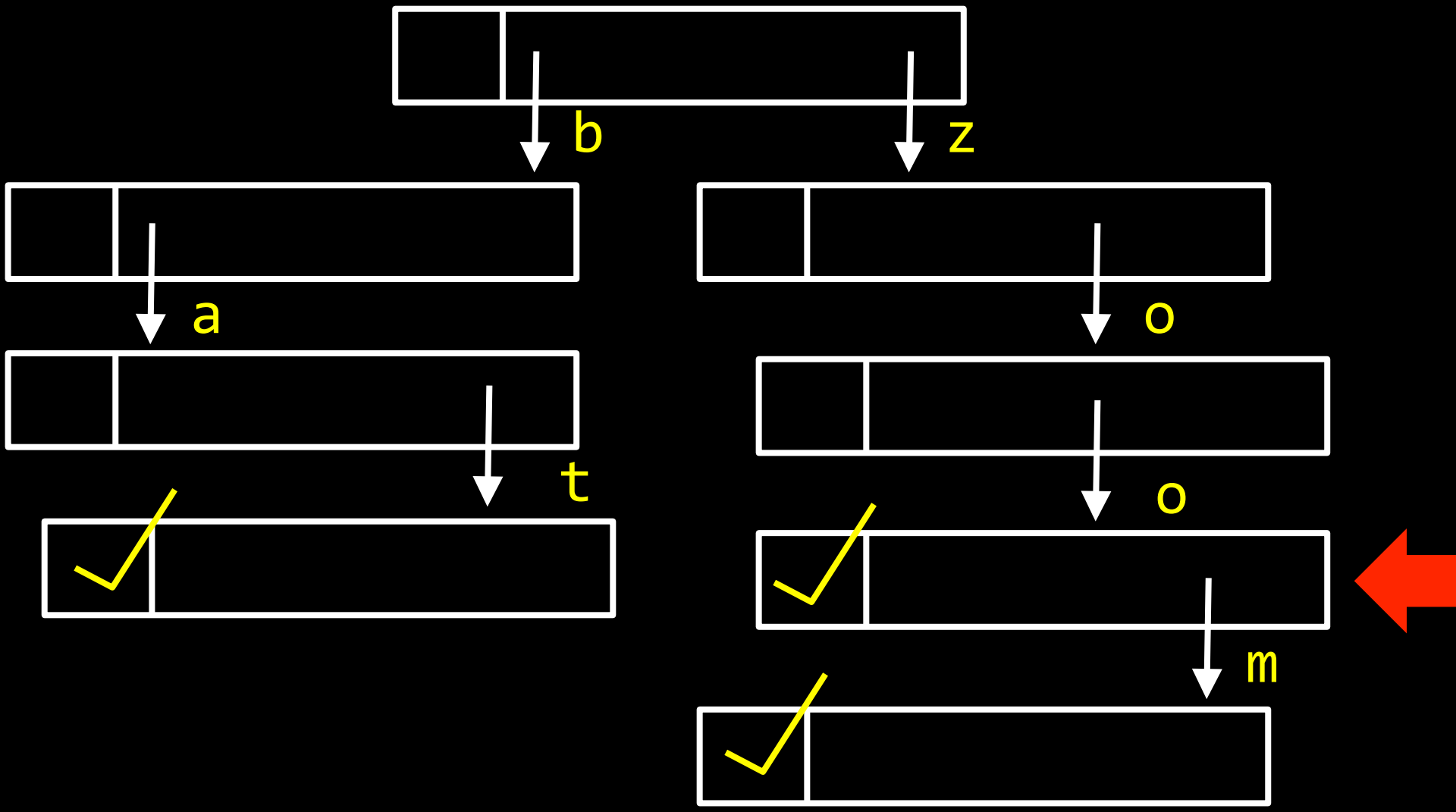

b
z
a
o
t
o
m

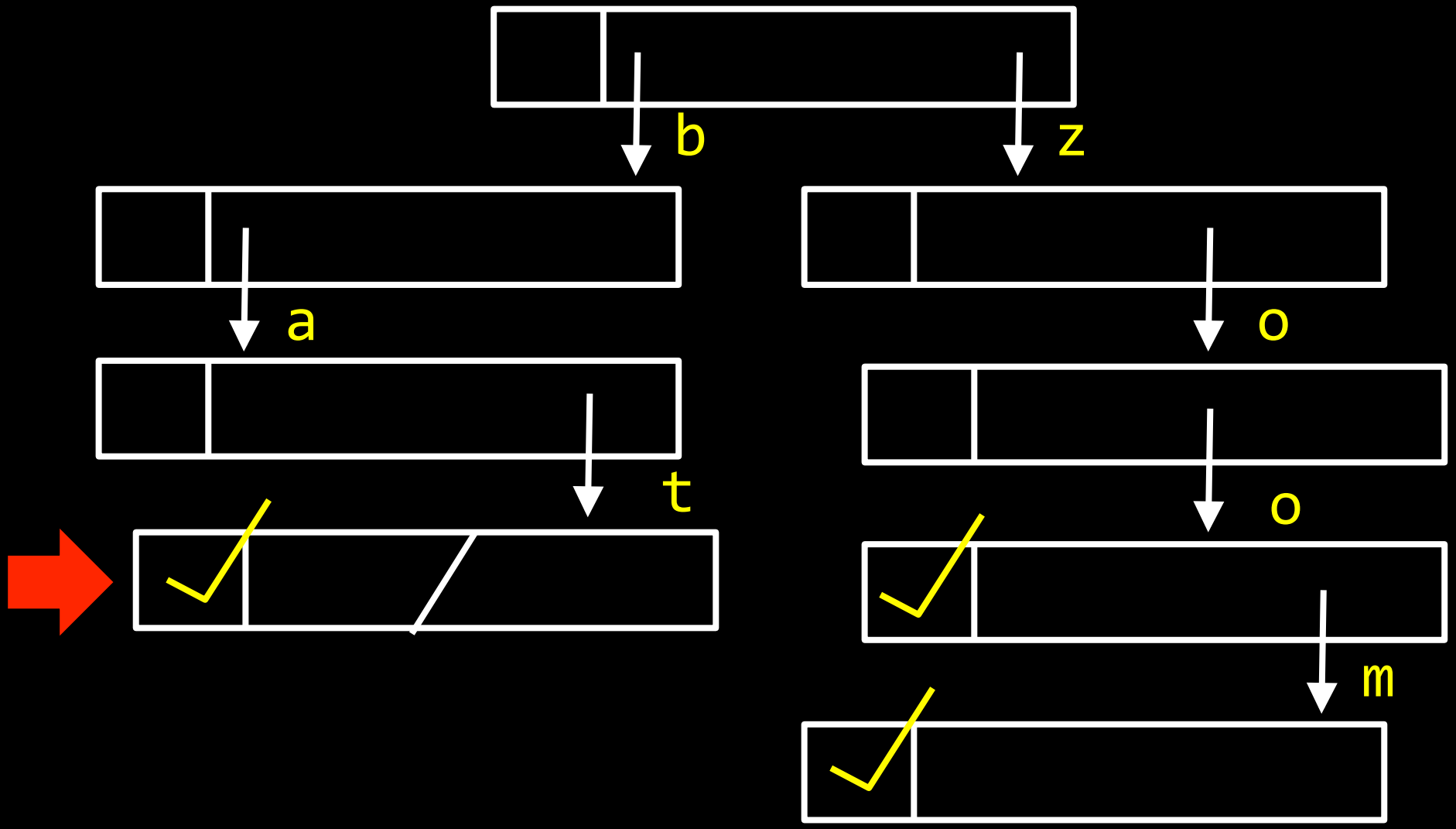

b
z
a
o
t
o
m

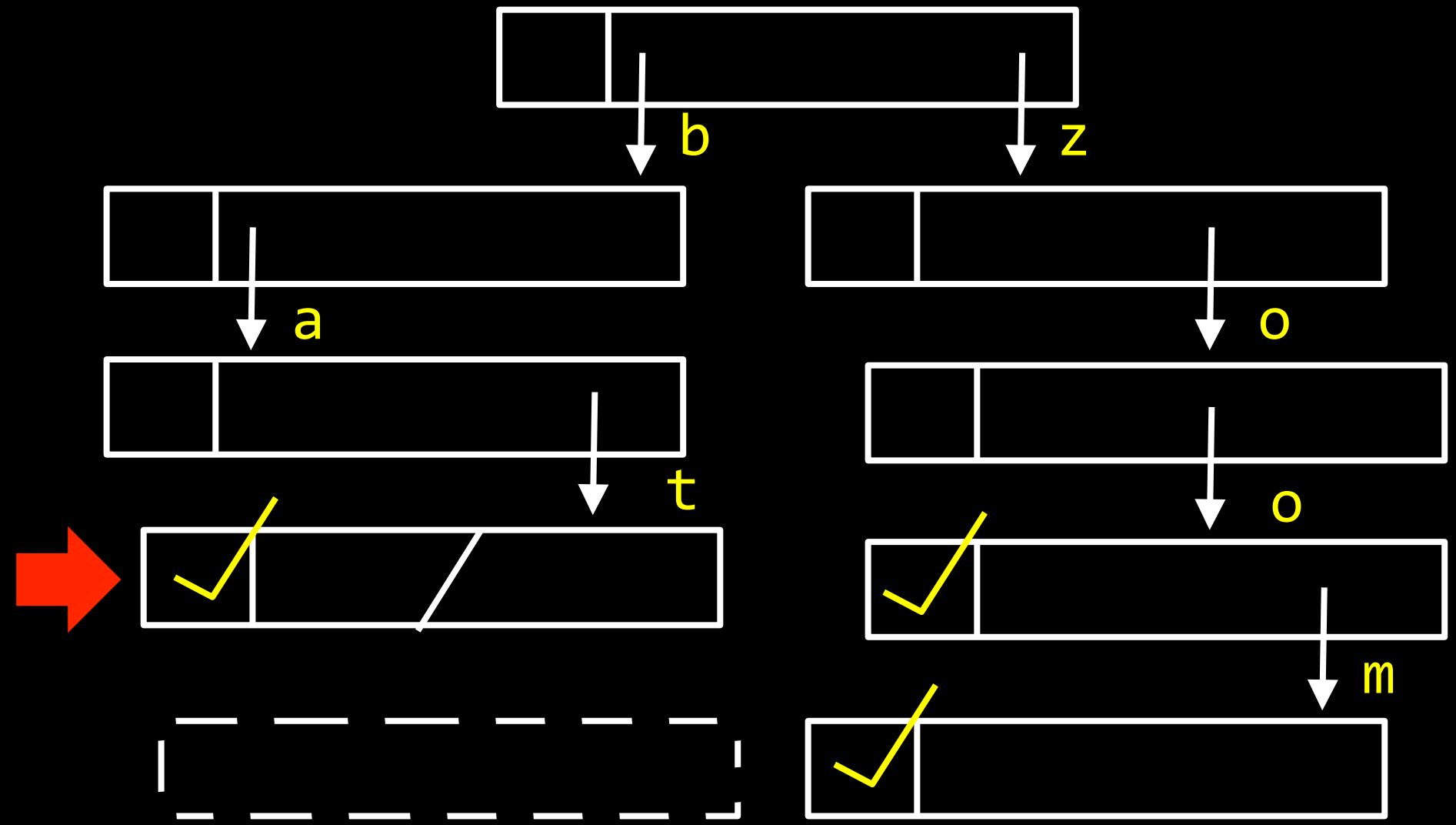

b
z
a
o
t
o
m

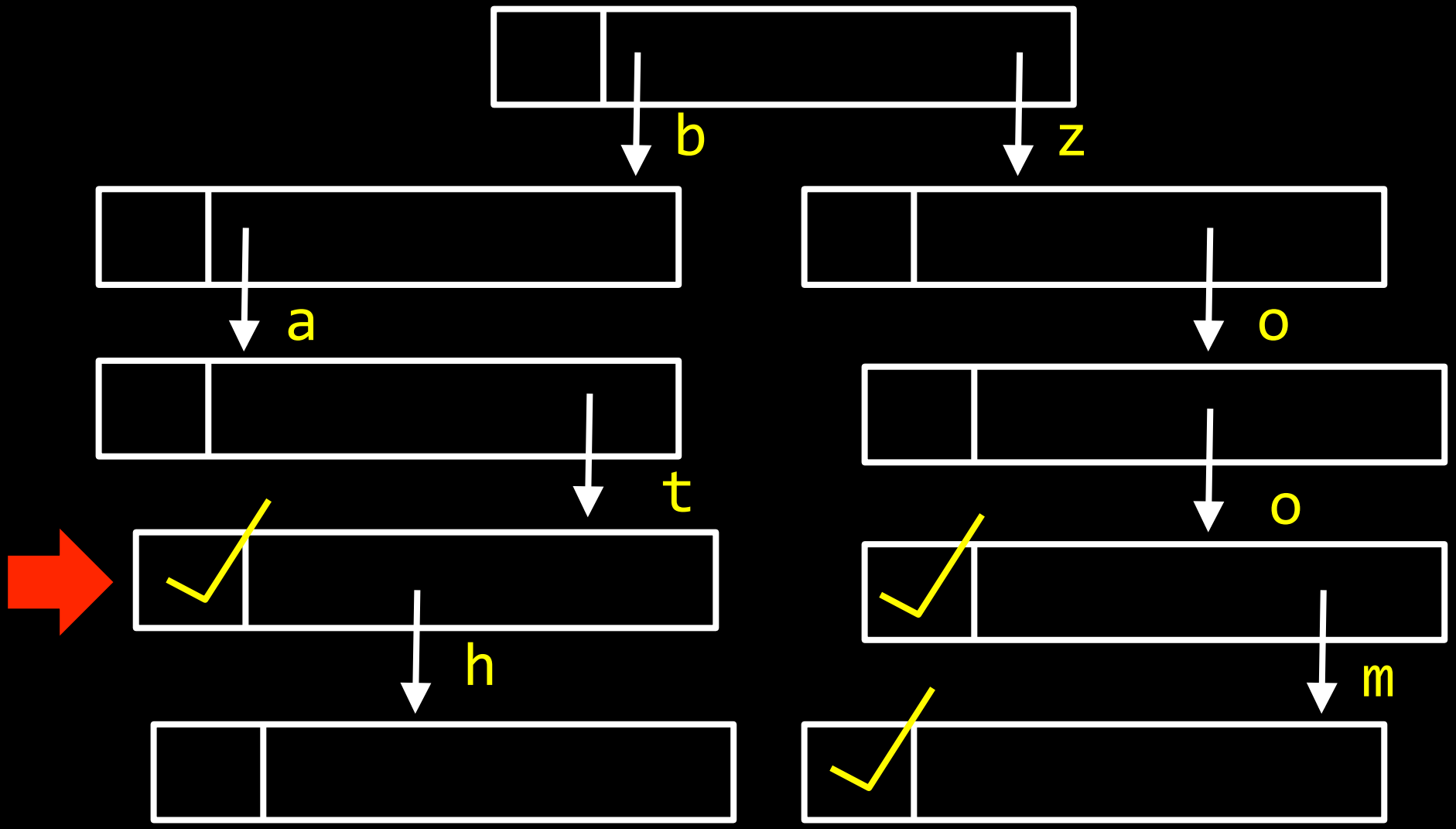

b
z
a
o
t
o
m

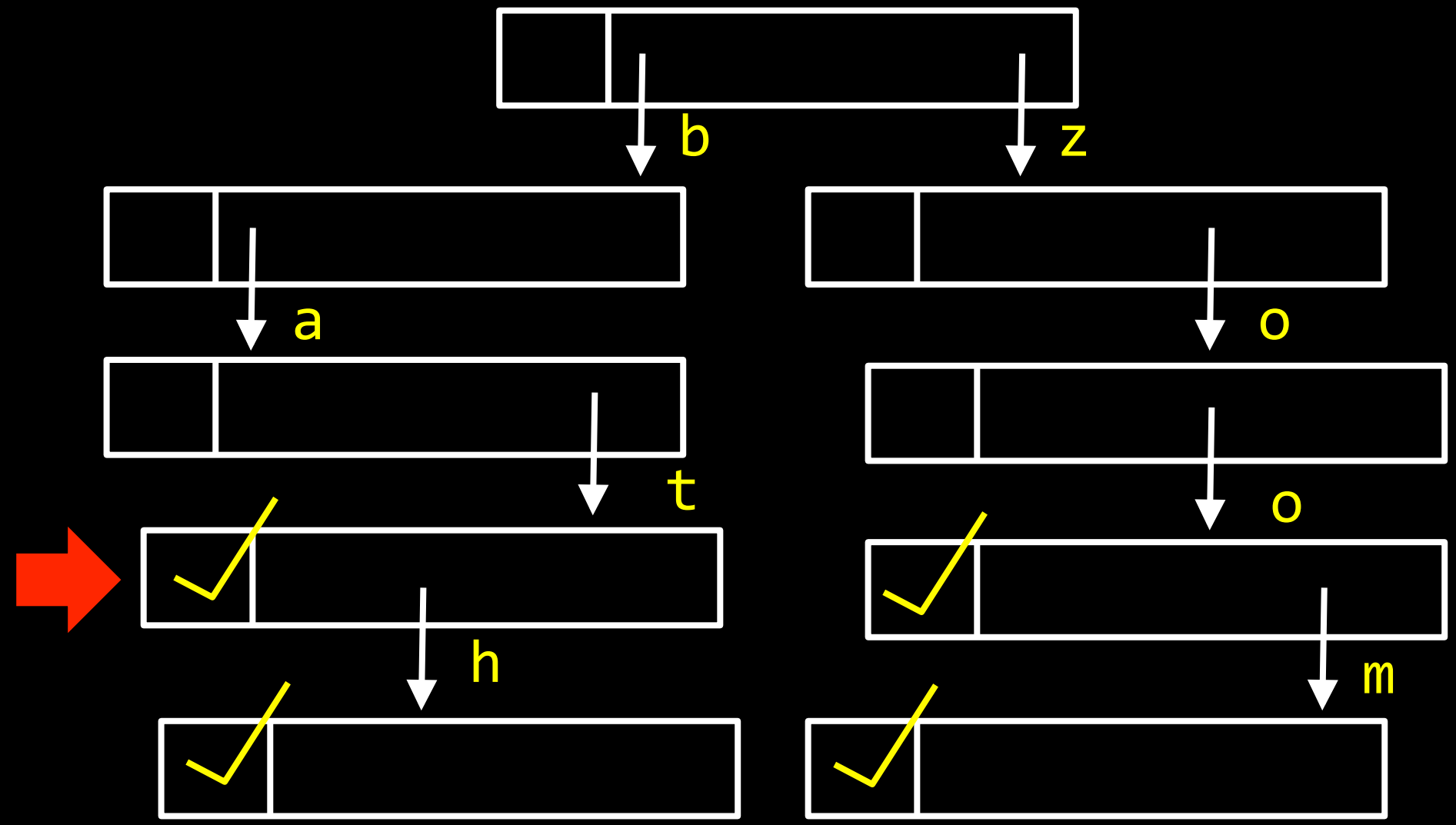

b
z
a
o
t
o
m


b
z
a
o
t
o
m

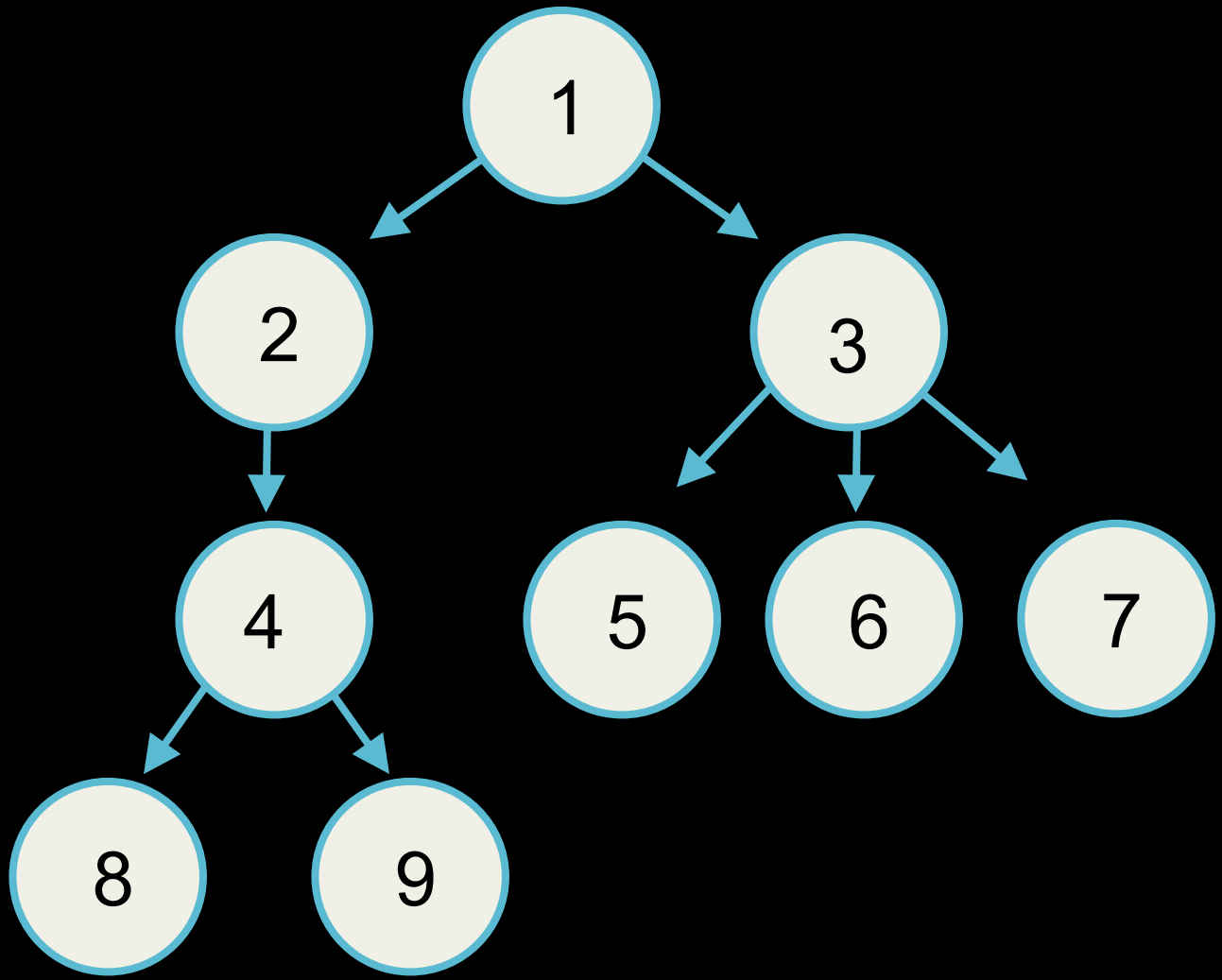

b
z
a
o
t
o
m

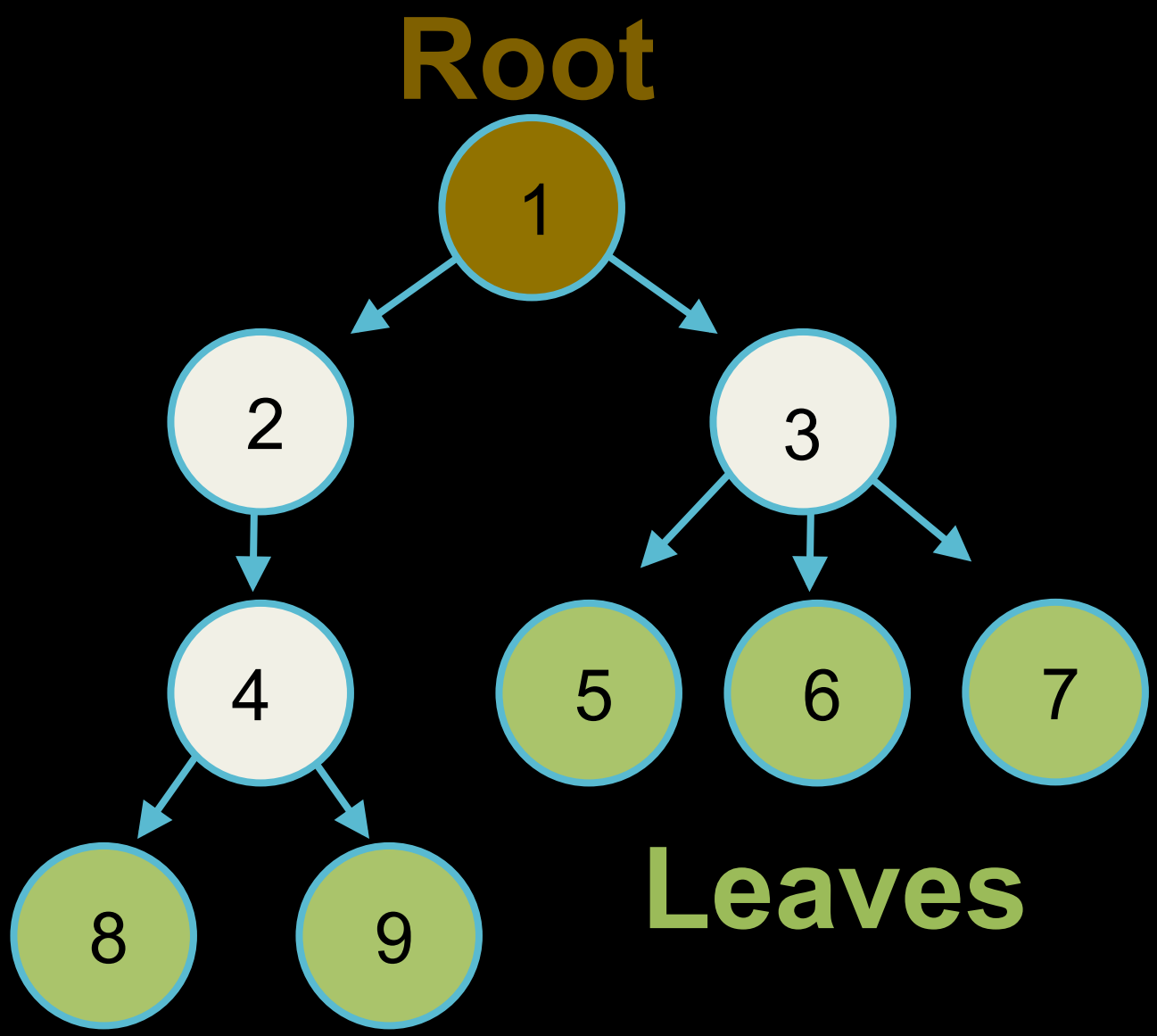

b
z
a
o
t
o
m


b
z
a
o
t
o
h
m

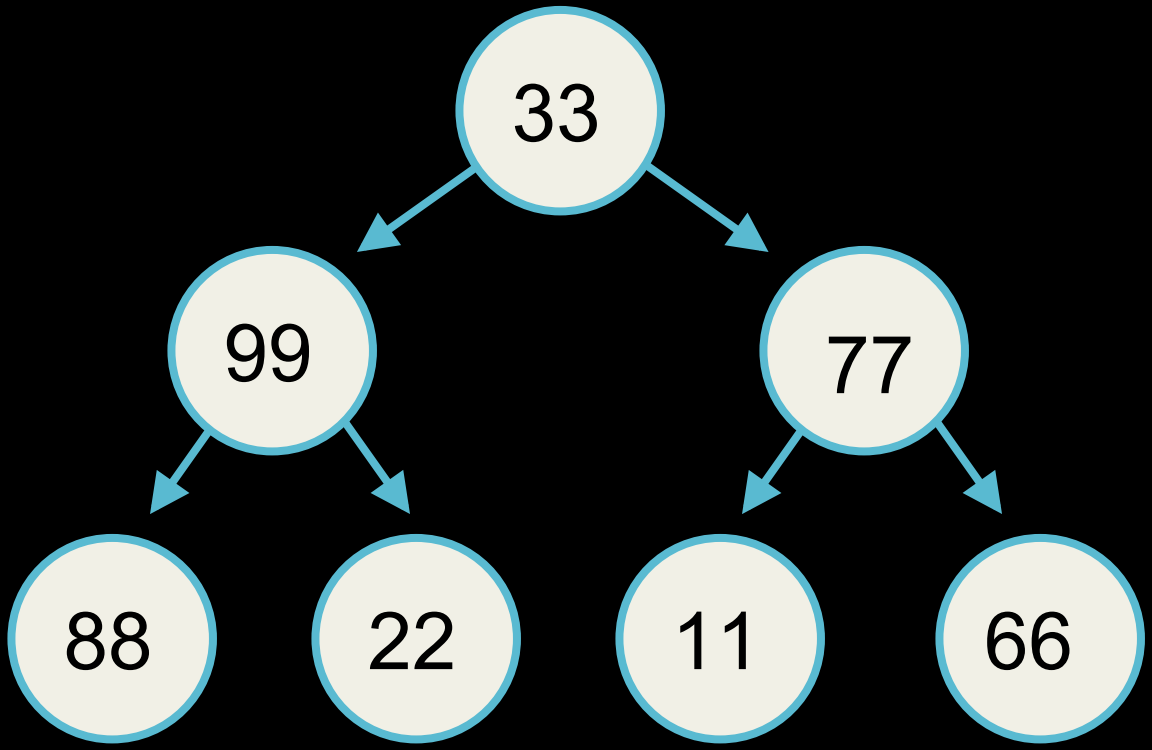

b
z
a
o
t
o
h
m

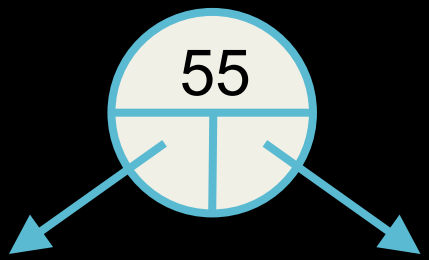
Tree




Root
1
2
3
4
5
6
7
8
9
Leaves

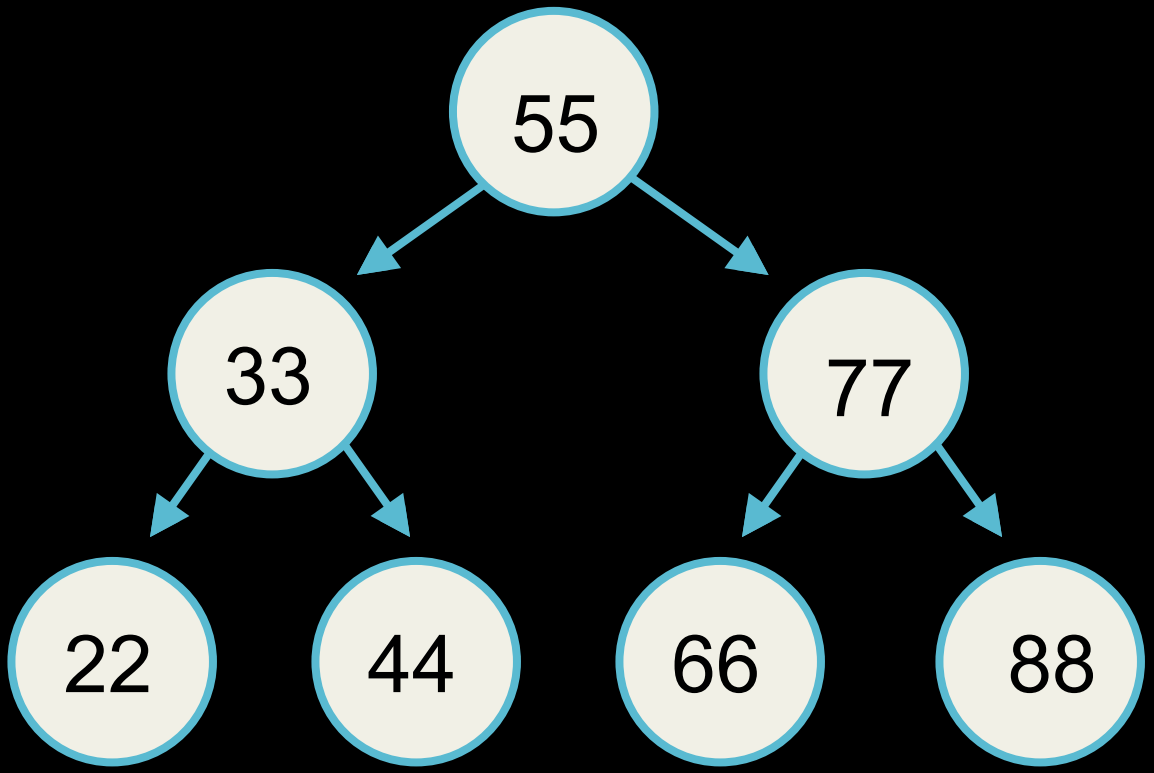
Binary Tree





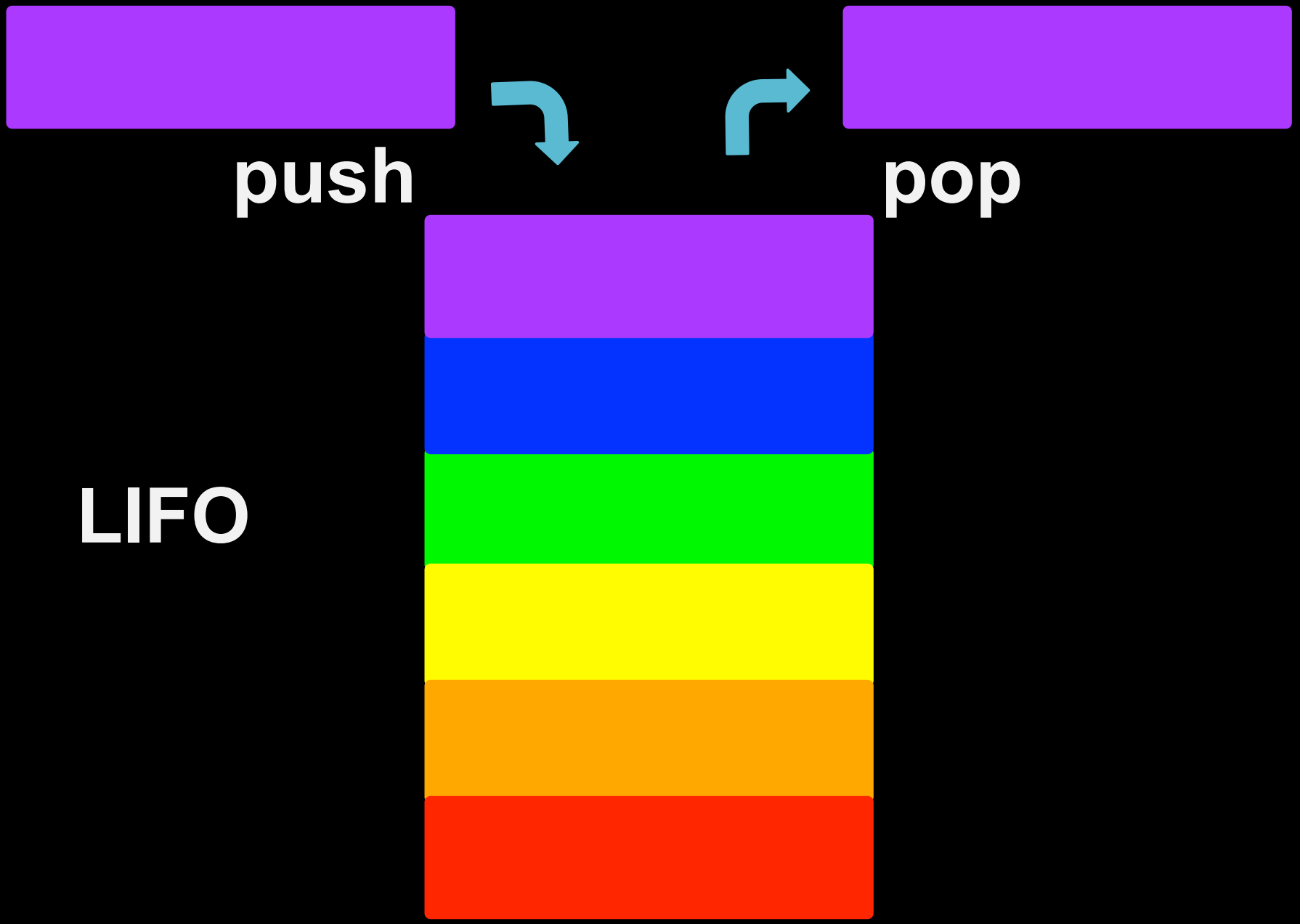

```
typedef struct node
{
     int n;
     struct node* left;
     struct node*
right;
}
node;
```

Binary Search Tree

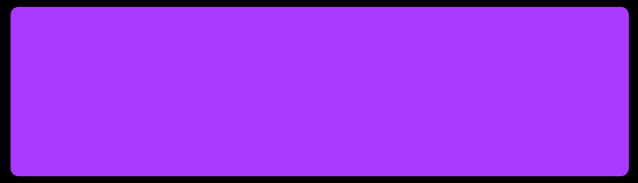


```
bool search(node* root, int val)
{
    if root is NULL
        return false.

    if root->n is val
        return true.

    if val is less than root->n
        search left child

    if val is greater than root->n
        search right child
}
```

Stacks

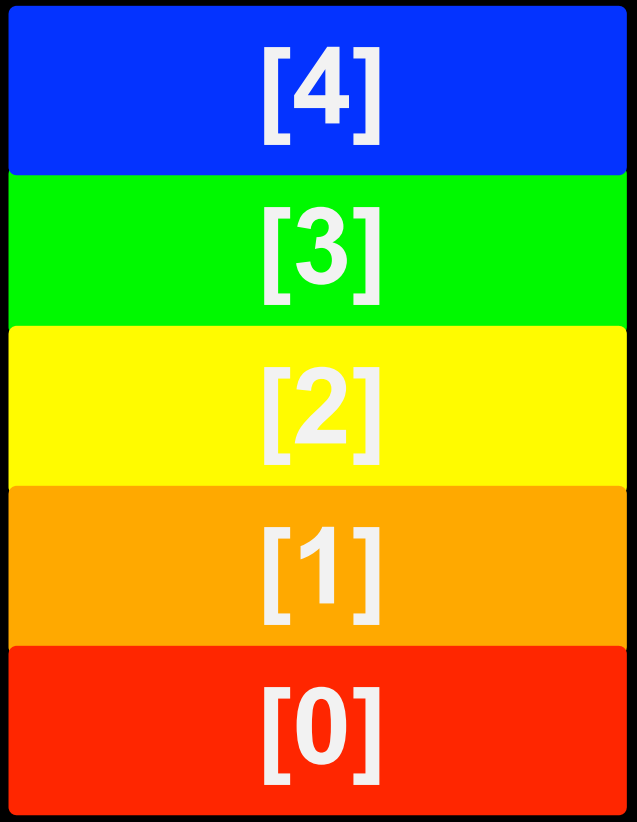

push
pop
LIFO

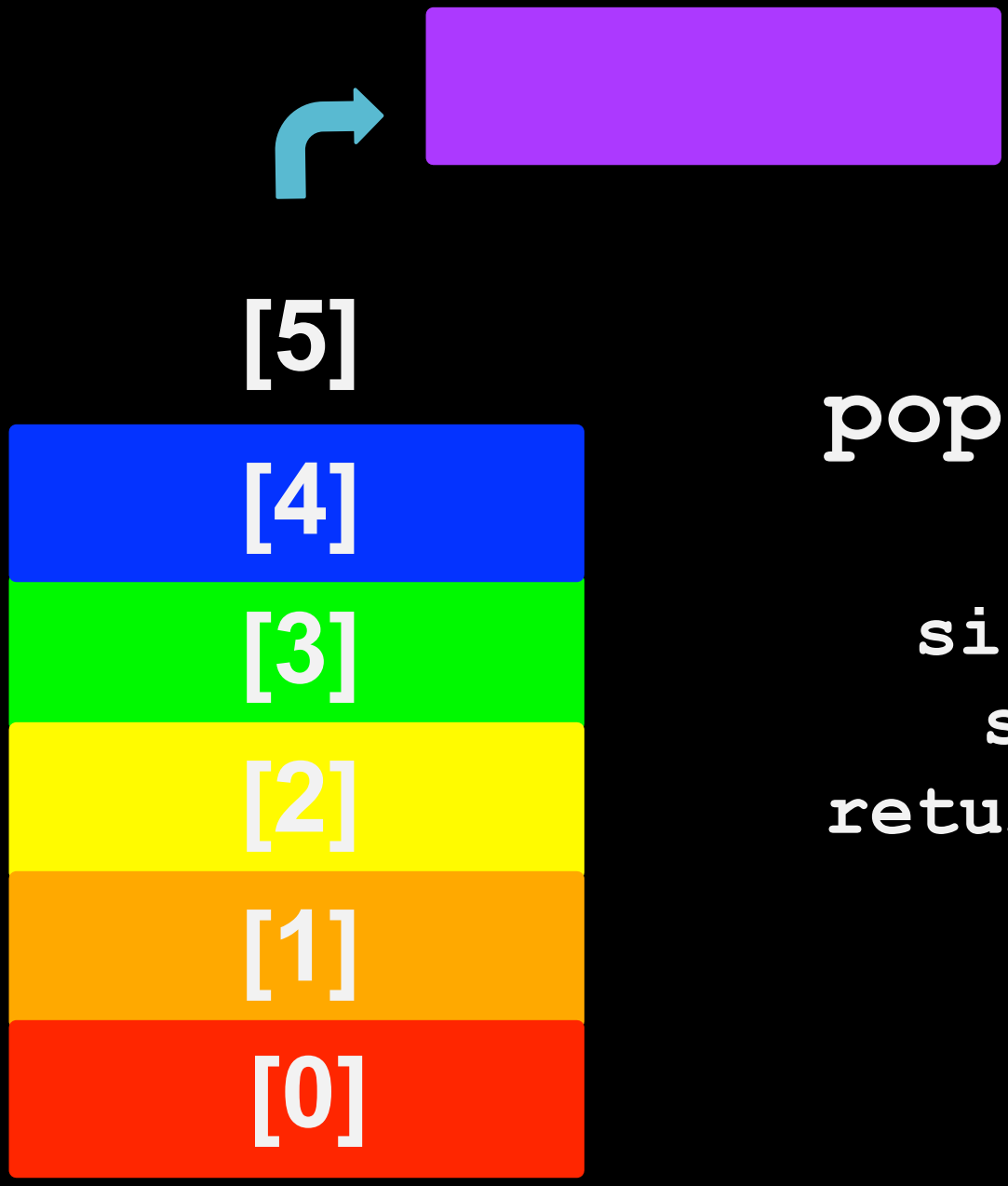
```
typedef struct
{
   char* strings[CAPACITY];
   int size;
}
stack;
```



`push` TODOs:

`size < CAPACITY?`
store element at
`[size]`
`size++`



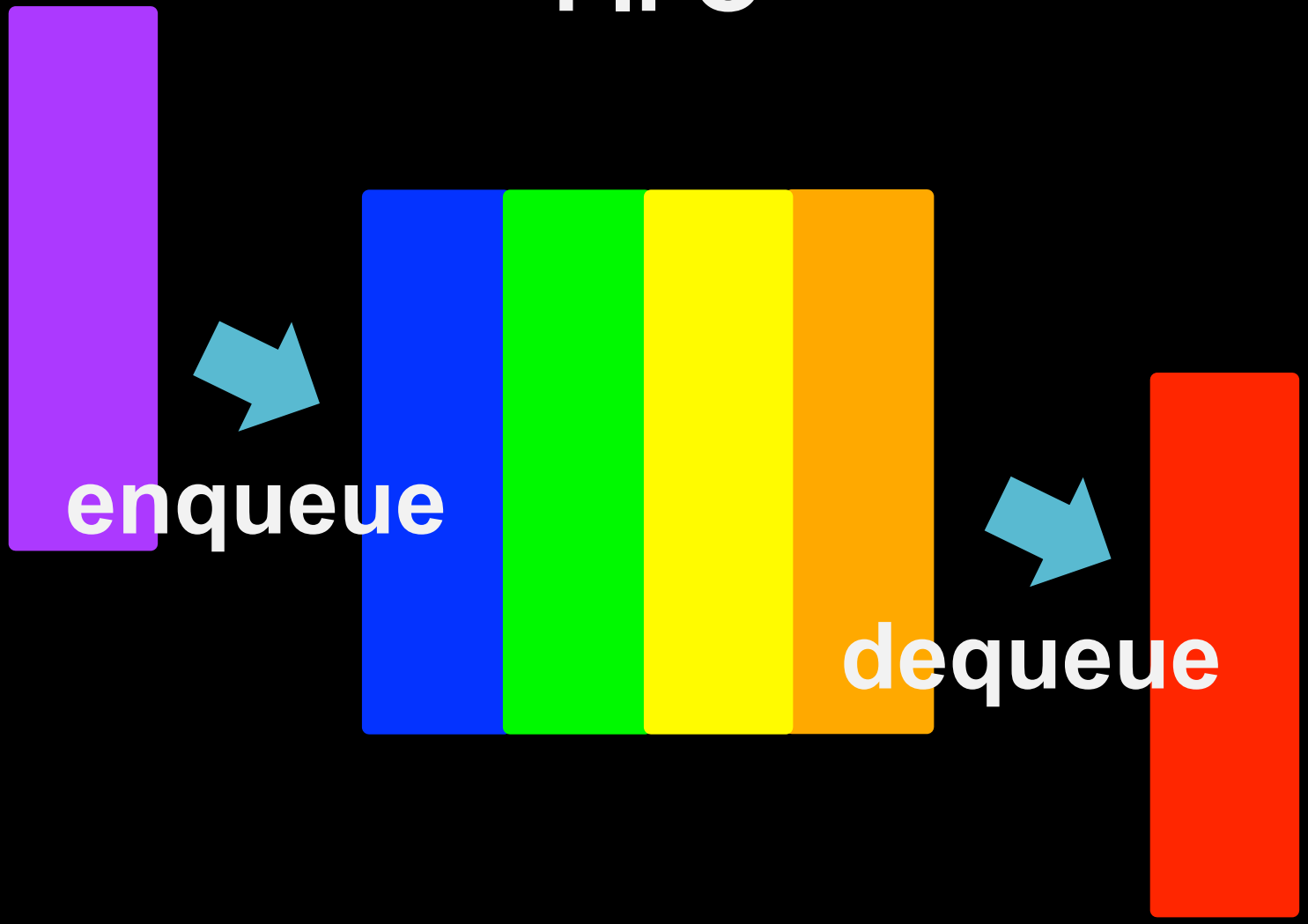


pop TODOs:

```
size > 0?
size--
return [size]
```

Queues

FIFO



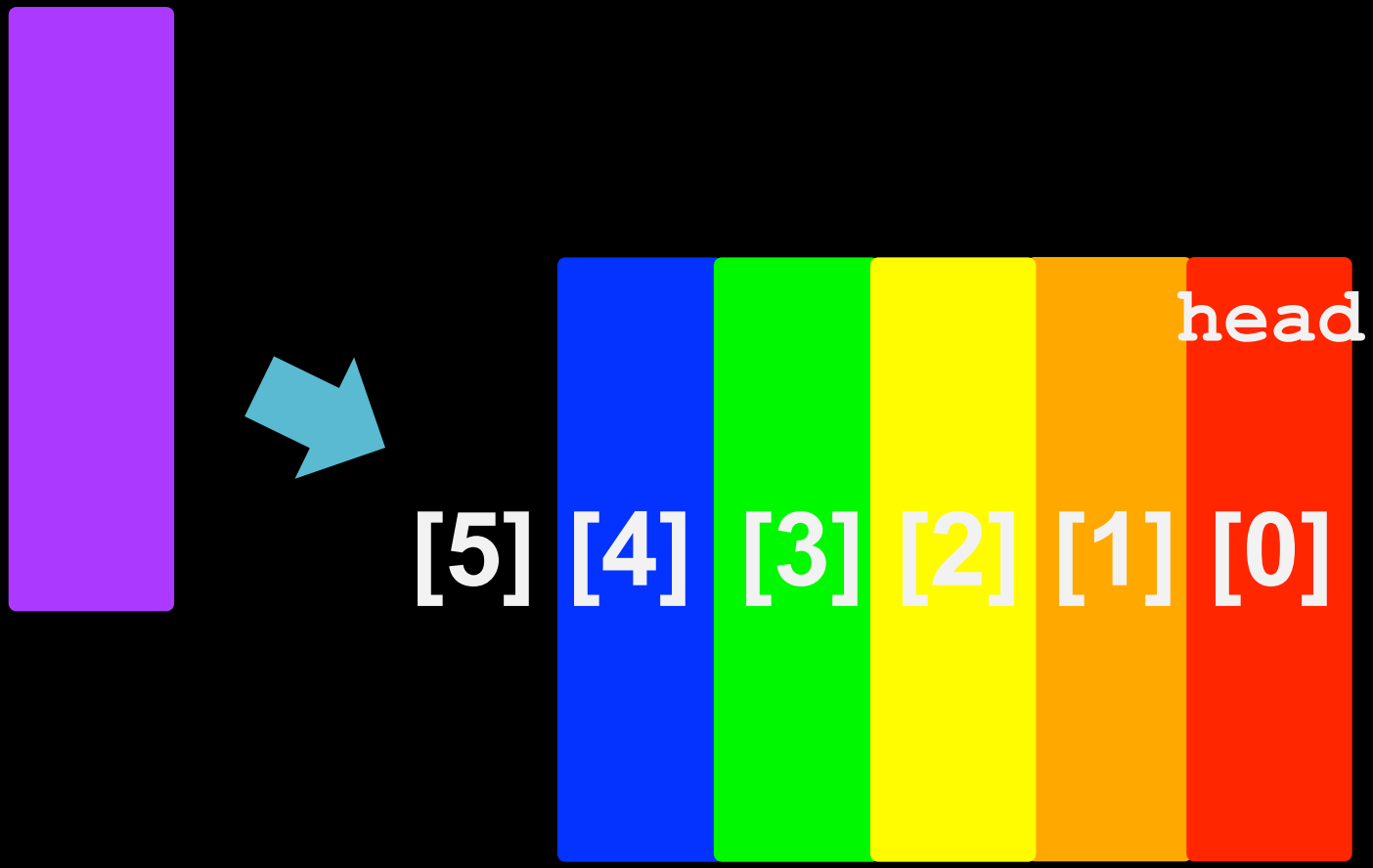
```
typedef struct
{
    int head;
    char* strings[CAPACITY];
    int size;
}
queue;
```

Enqueue TODOs:

`size < CAPACITY?`

store at tail

`size++`



Dequeue TODOs:

`size > 0?`

move `head`

`size--`

return element