Singly-Linked Lists

Singly-Linked Lists

- So far in the course, we've only had one kind of data structure for representing collections of like values.
    - `structs`, recall, give us "containers" for holding variables of different data types, typically.

- Arrays are great for element lookup, but unless we want to insert at the very end of the array, inserting elements is quite costly – remember insertion sort?

Singly-Linked Lists

- Arrays also suffer from a great inflexibility – what happens if we need a larger array than we thought?

- Through clever use of pointers, dynamic memory allocation, and `struct`s, we can put those two pieces together to develop a new kind of data structure that gives us the ability to grow and shrink a collection of like values to fit our needs.

Singly-Linked Lists

- We call this combination of elements, when used in this way, a **linked list**.

- 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 Lists

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

Singly-Linked Lists

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

Singly-Linked Lists

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

Singly-Linked Lists

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

Singly-Linked Lists

- In order to work with linked lists effectively, there are a number of operations that we need to understand:

1. Create a linked list when it doesn’t already exist.
2. Search through a linked list to find an element.
3. Insert a new node into the linked list.
4. Delete a single element from a linked list.
5. Delete an entire linked list.

Singly-Linked Lists

- Create a linked list.

```
sllnode* create(VALUE val);
```

Singly-Linked Lists

- Create a linked list.

```
sllnode* create(VALUE val);
```

- Steps involved:
  a. Dynamically allocate space for a new `sllnode`.
  b. Check to make sure we didn’t run out of memory.
  c. Initialize the node’s `val` field.
  d. Initialize the node’s `next` field.
  e. Return a pointer to the newly created `sllnode`.

Singly-Linked Lists

```
sllnode* new = create(6);
```

a. Dynamically allocate space for a new `sllnode`.
b. Check to make sure we didn't run out of memory.
c. Initialize the node's `val` field.
d. Initialize the node's `next` field.
e. Return a pointer to the newly created `sllnode`.

Singly-Linked Lists

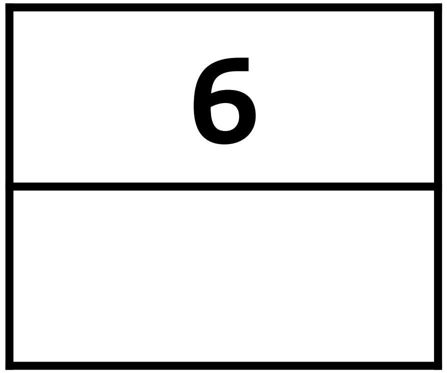
```
sllnode* new = create(6);
```

a. Dynamically allocate space for a new `sllnode`.
b. Check to make sure we didn't run out of memory.
c. Initialize the node's `val` field.
d. Initialize the node's `next` field.
e. Return a pointer to the newly created `sllnode`.

Singly-Linked Lists

```
sllnode* new = create(6);
```

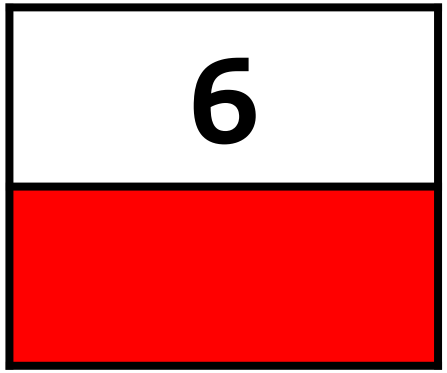
a. Dynamically allocate space for a new `sllnode`.
b. Check to make sure we didn't run out of memory.
c. Initialize the node's `val` field.
d. Initialize the node's `next` field.
e. Return a pointer to the newly created `sllnode`.



Singly-Linked Lists

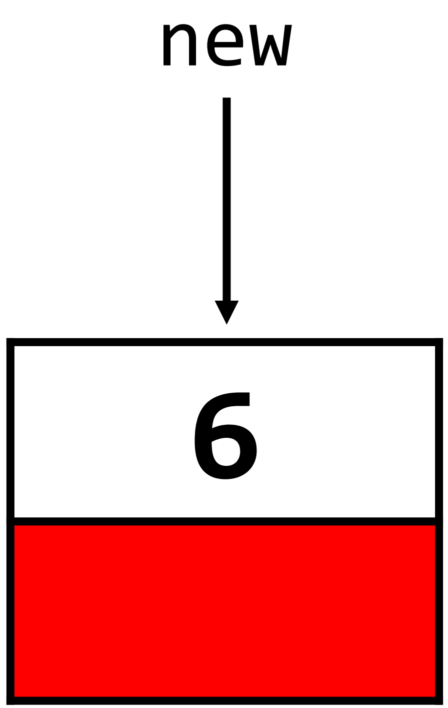
```
sllnode* new = create(6);
```

a. Dynamically allocate space for a new `sllnode`.
b. Check to make sure we didn't run out of memory.
c. Initialize the node's `val` field.
d. Initialize the node's `next` field.
e. Return a pointer to the newly created `sllnode`.



Singly-Linked Lists

```
sllnode* new = create(6);
```



a. Dynamically allocate space for a new `sllnode`.
b. Check to make sure we didn't run out of memory.
c. Initialize the node's `val` field.
d. Initialize the node's `next` field.
e. Return a pointer to the newly created `sllnode`.

Singly-Linked Lists

- Search through a linked list to find an element.

```
bool find(sllnode* head, VALUE val);
```

Singly-Linked Lists

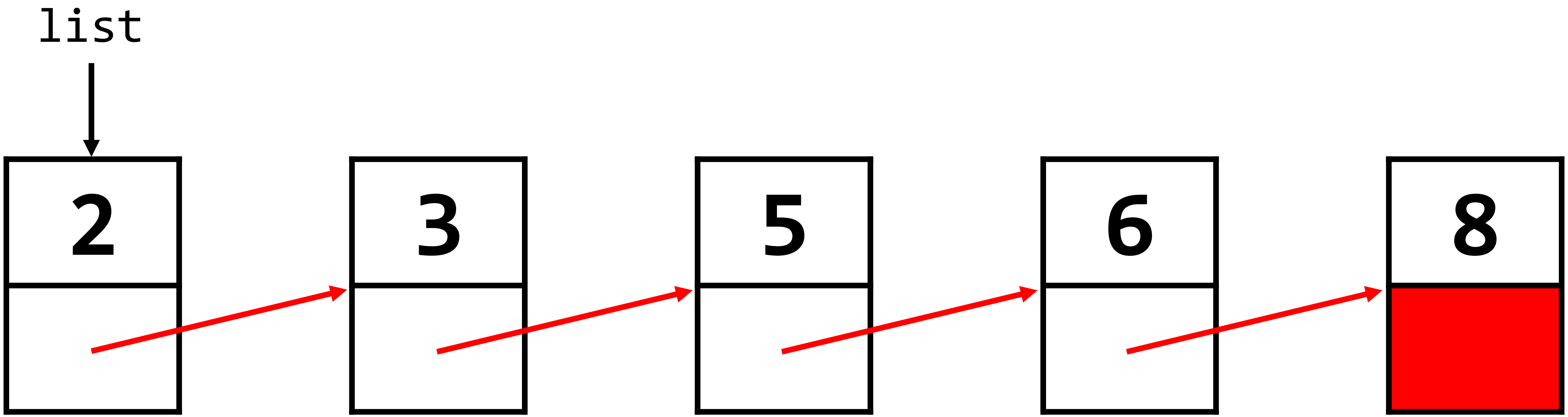
- Search through a linked list to find an element.

```
bool find(sllnode* head, VALUE val);
```

- Steps involved:
  a. Create a traversal pointer pointing to the list's head.
  b. If the current node's `val` field is what we're looking for, report success.
  c. If not, set the traversal pointer to the next pointer in the list and go back to step b.
  d. If you've reached the end of the list, report failure.

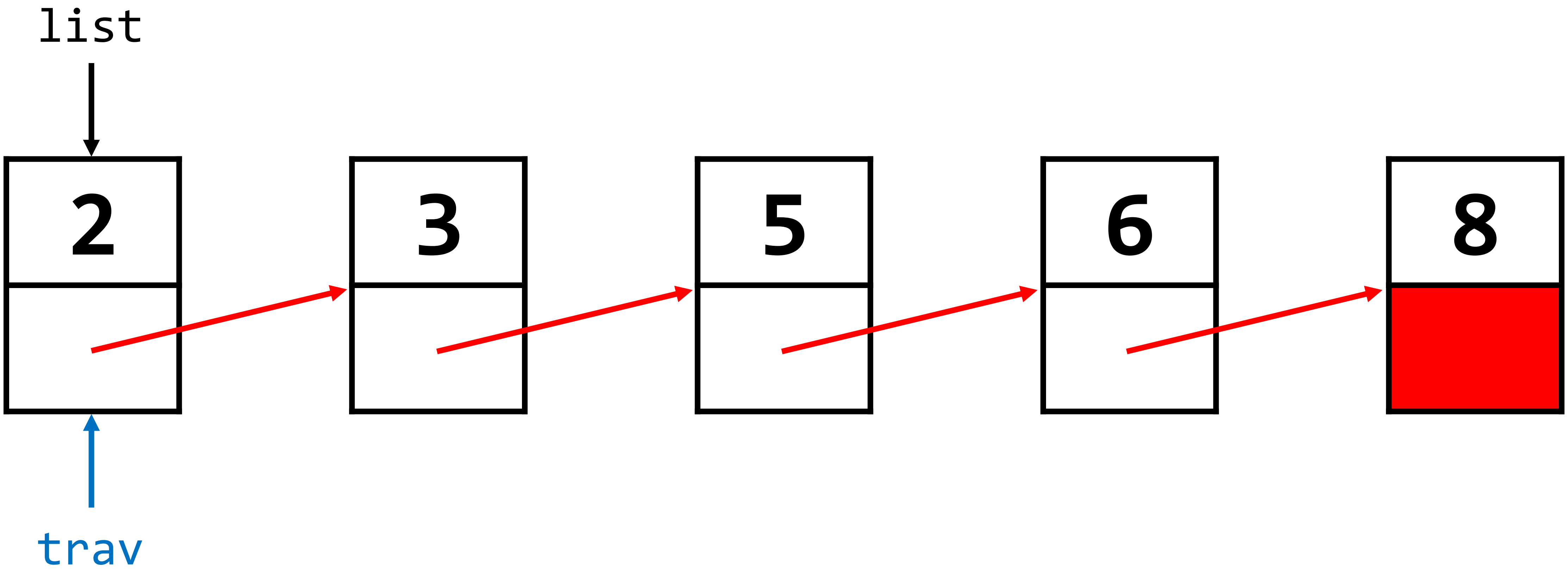
Singly-Linked Lists

```
bool exists = find(list, 6);
```



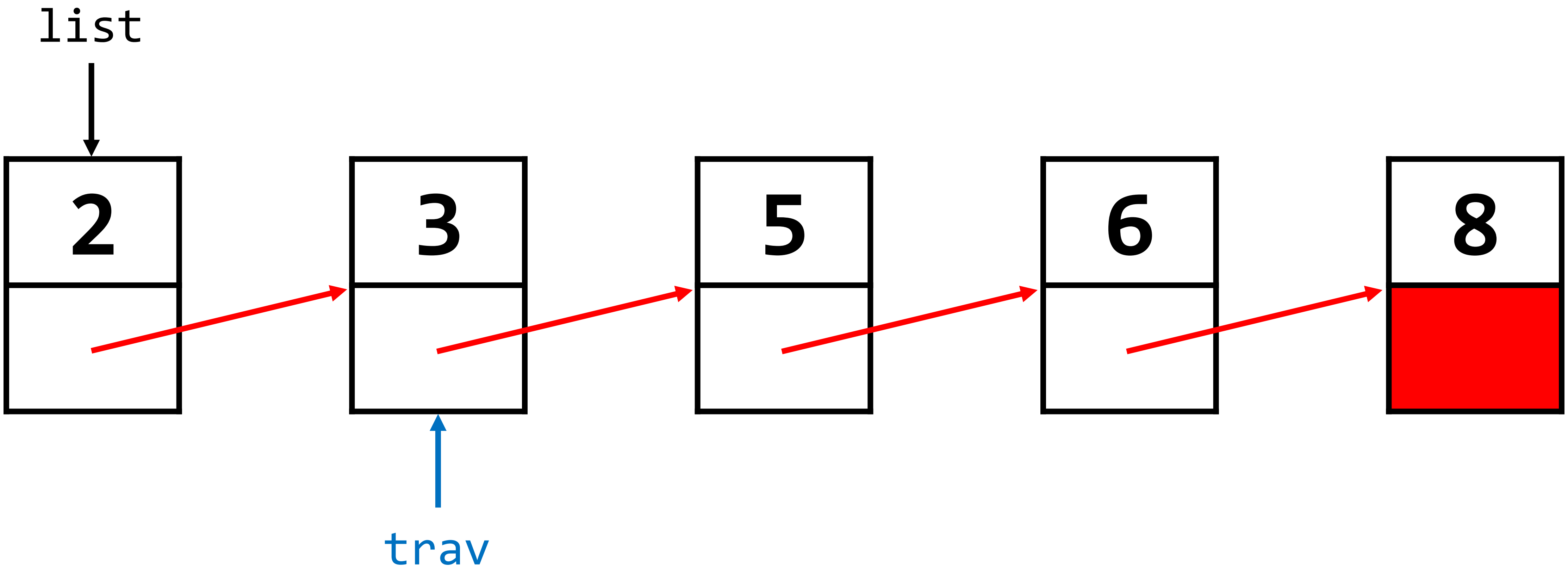
Singly-Linked Lists

```
bool exists = find(list, 6);
```



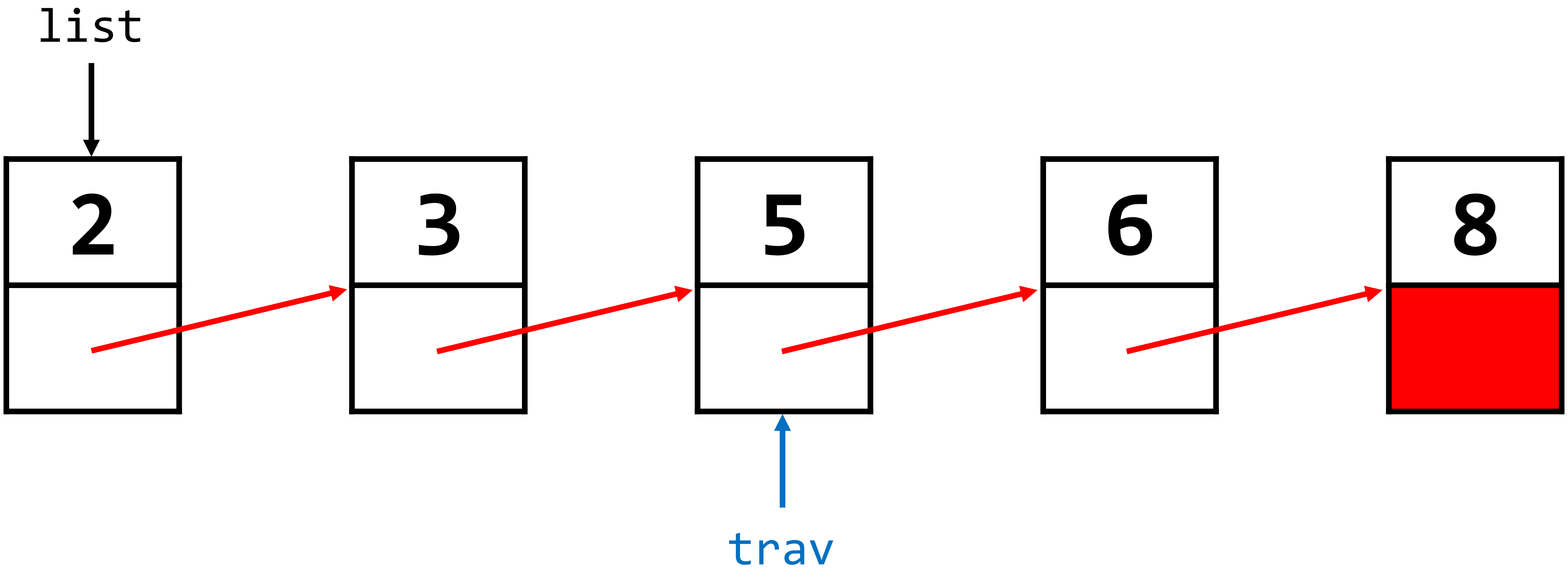
Singly-Linked Lists

```
bool exists = find(list, 6);
```



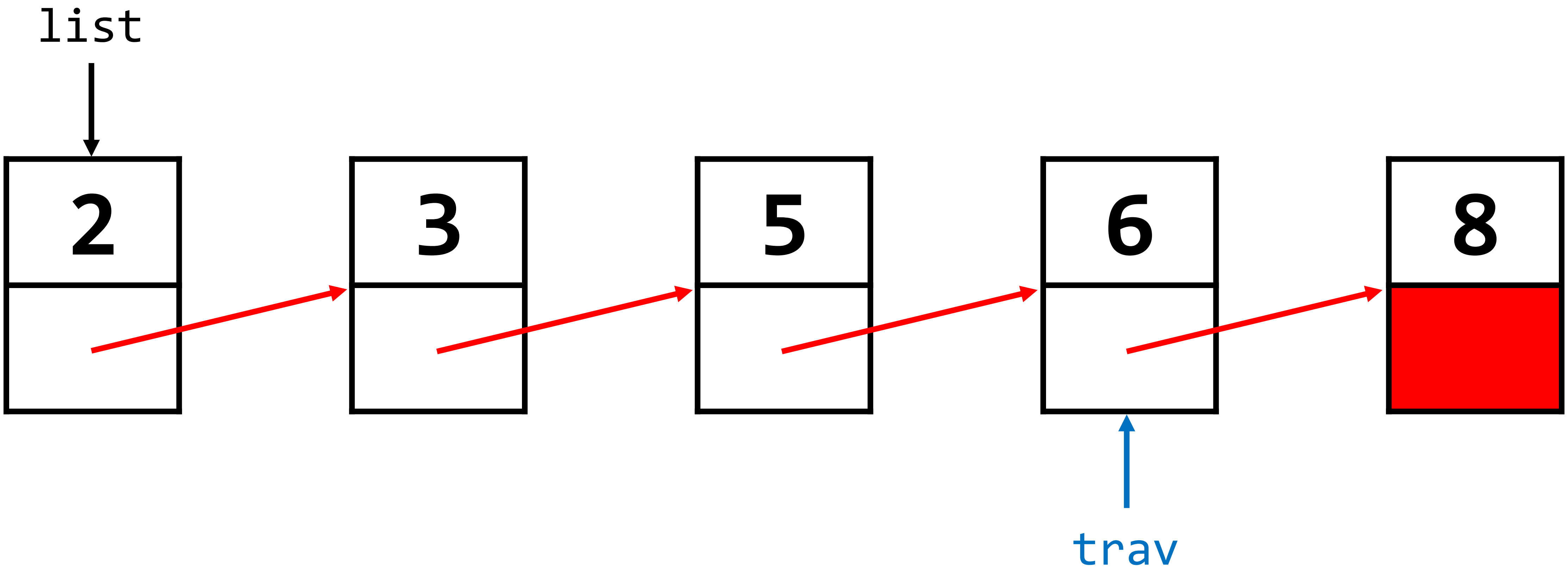
Singly-Linked Lists

```
bool exists = find(list, 6);
```



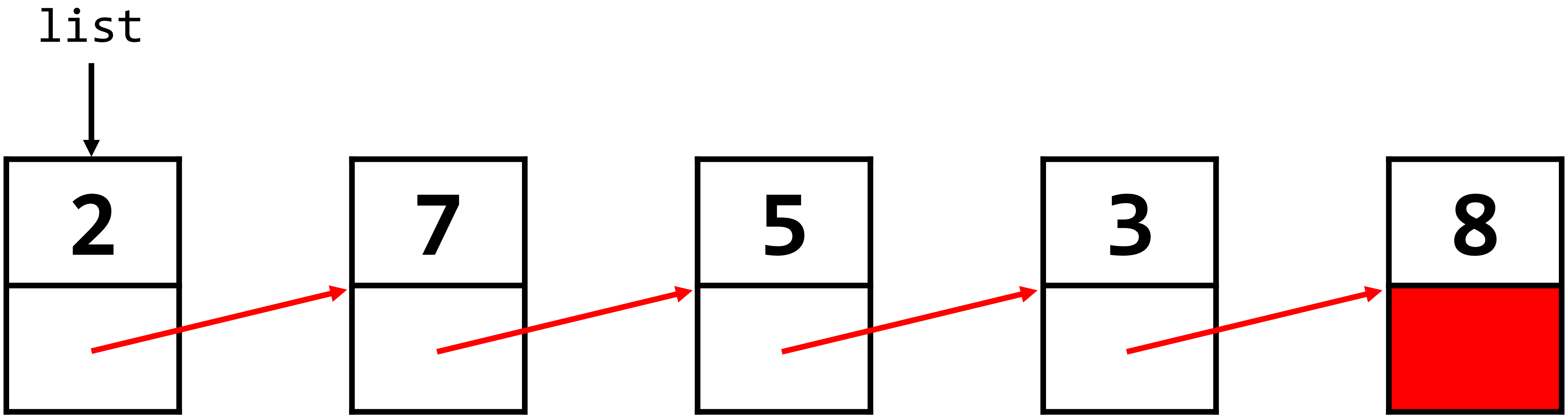
Singly-Linked Lists

`bool exists = find(list, 6);`



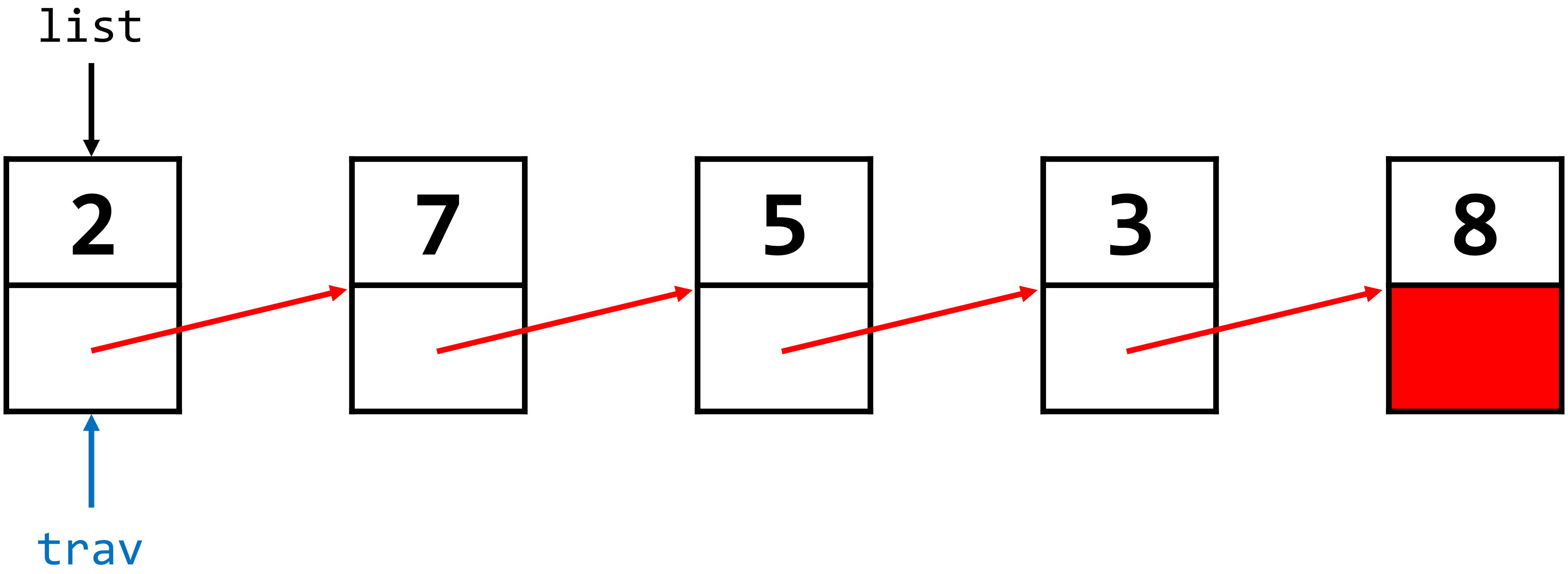
Singly-Linked Lists

```
bool exists = find(list, 6);
```



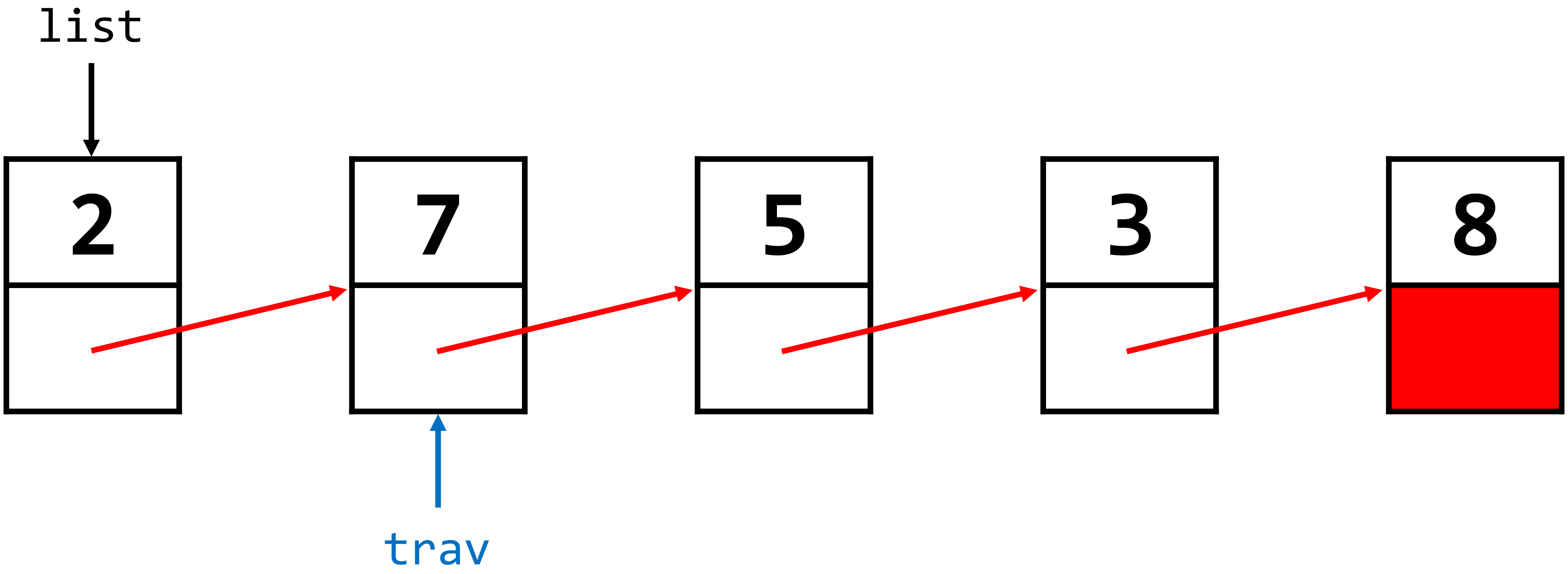
Singly-Linked Lists

```
bool exists = find(list, 6);
```



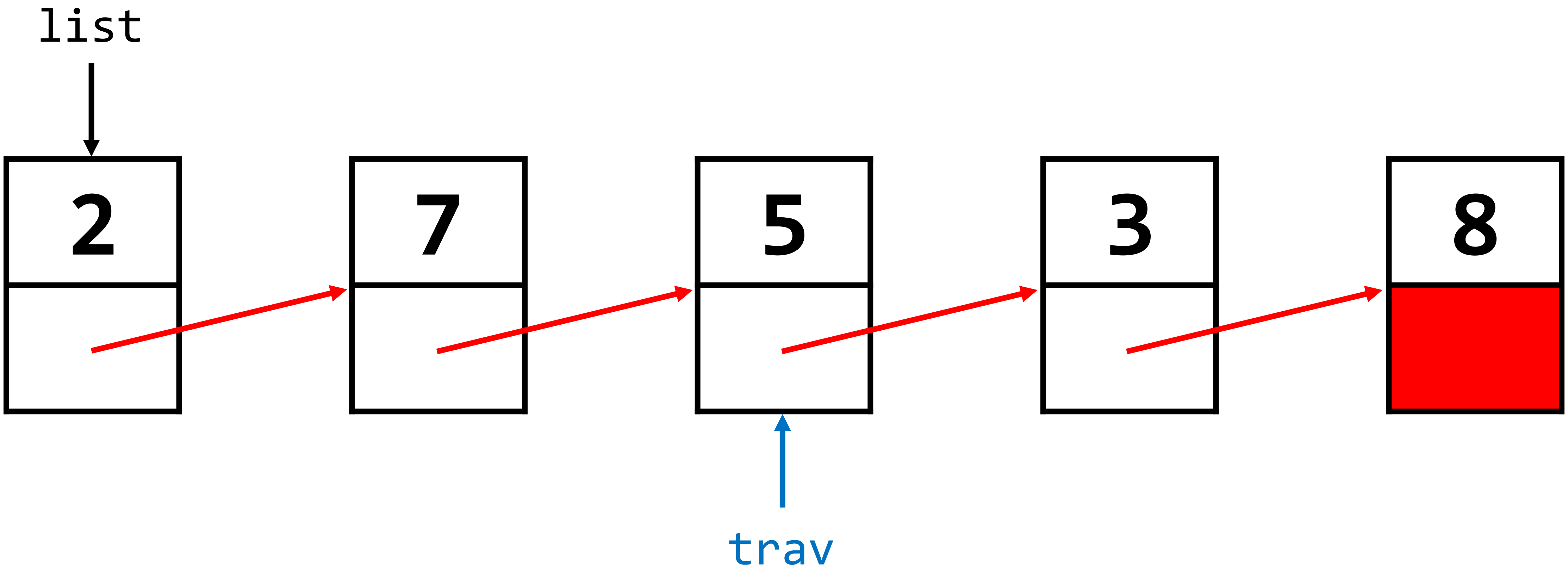
Singly-Linked Lists

```
bool exists = find(list, 6);
```



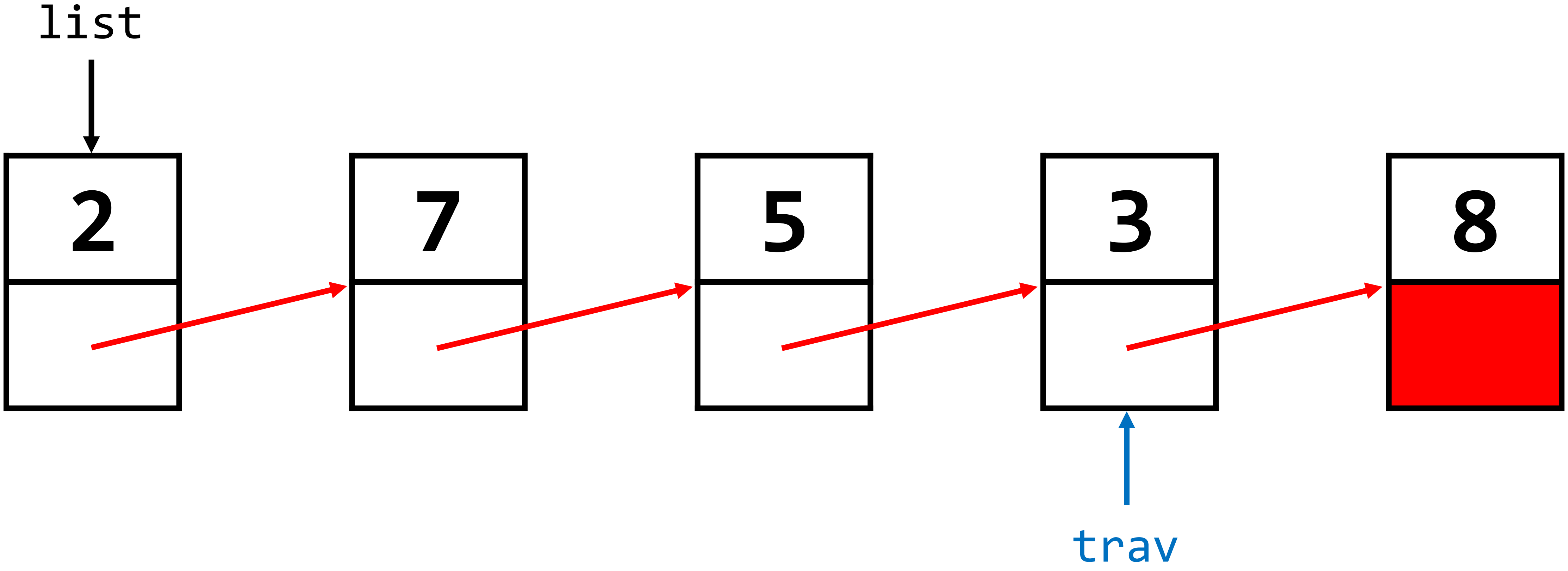
Singly-Linked Lists

```
bool exists = find(list, 6);
```



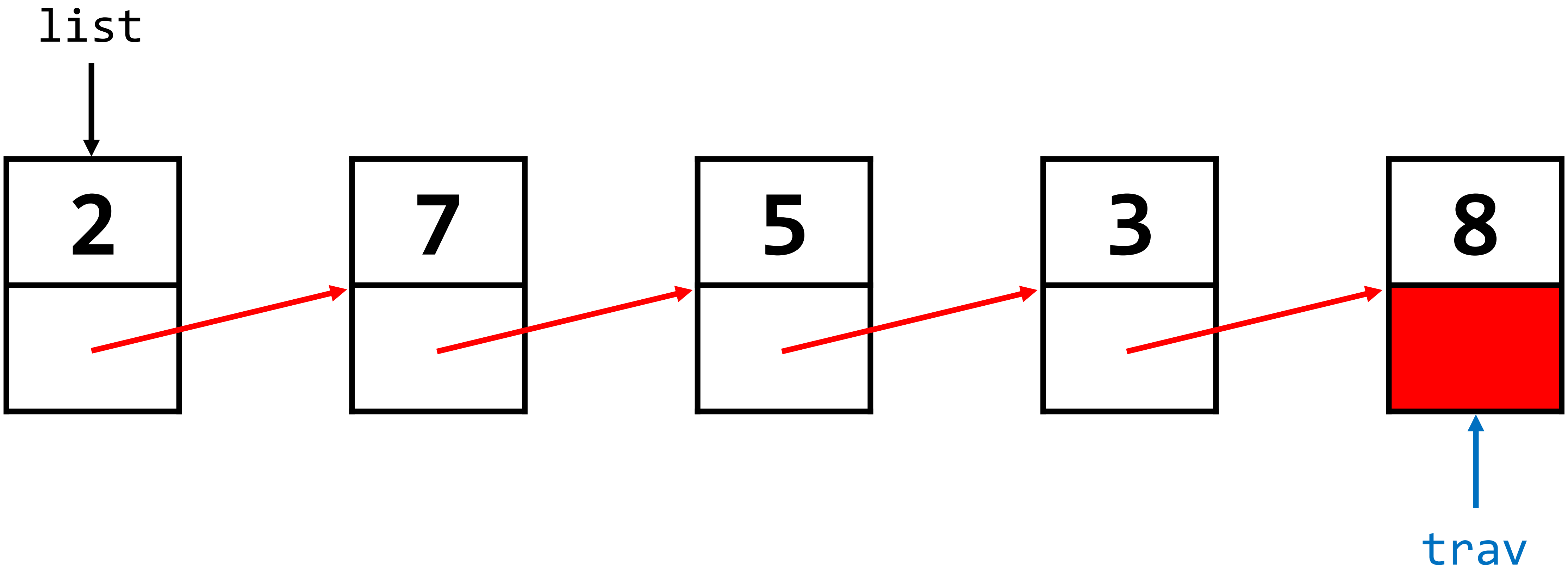
Singly-Linked Lists

`bool exists = find(list, 6);`



Singly-Linked Lists

`bool exists = find(list, 6);`



Singly-Linked Lists

- Insert a new node into the linked list.

```
sllnode* insert(sllnode* head, VALUE val);
```

Singly-Linked Lists

- Insert a new node into the linked list.

```
sllnode* insert(sllnode* head, VALUE val);
```

- Steps involved:
  a. Dynamically allocate space for a new `sllnode`.
  b. Check to make sure we didn't run out of memory.
  c. Populate and insert the node at the beginning of the linked list.
  d. Return a pointer to the new head of the linked list.

Singly-Linked Lists

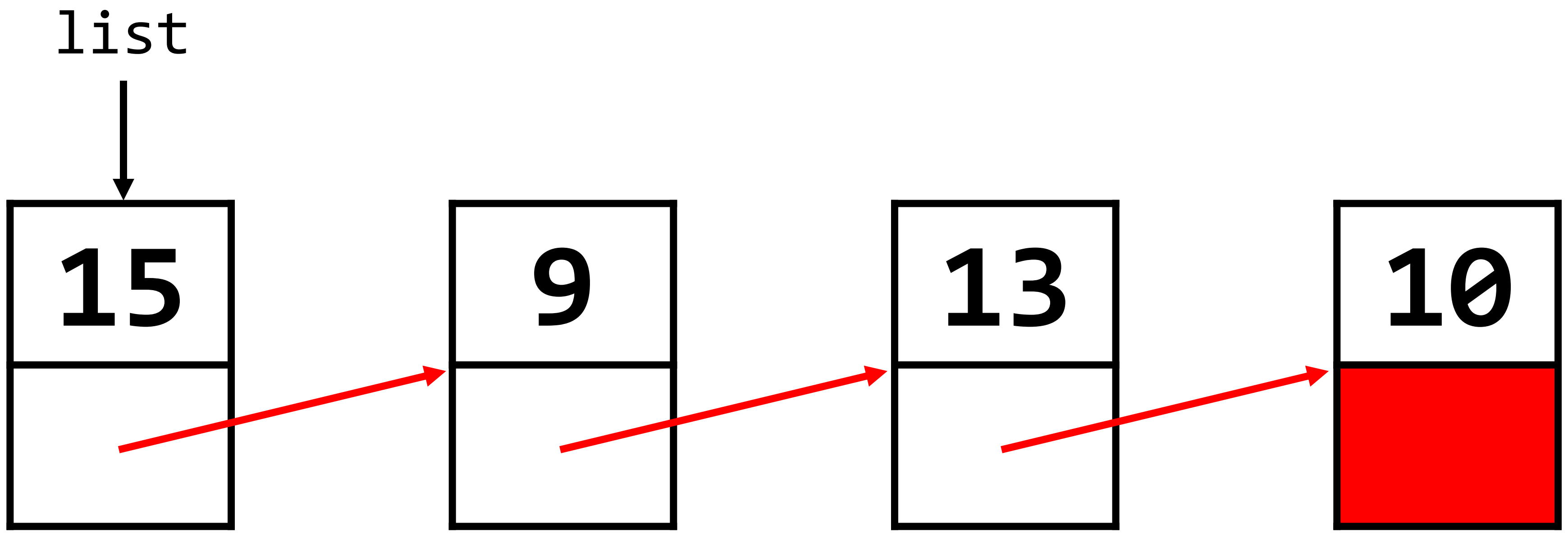
- Insert a new node into the linked list.

```
sllnode* insert(sllnode* head, VALUE val);
```

- Steps involved:
  a. Dynamically allocate space for a new `sllnode`.
  b. Check to make sure we didn’t run out of memory.
  c. Populate and insert the node at the beginning of the linked list.
  d. Return a pointer to the new head of the linked list.

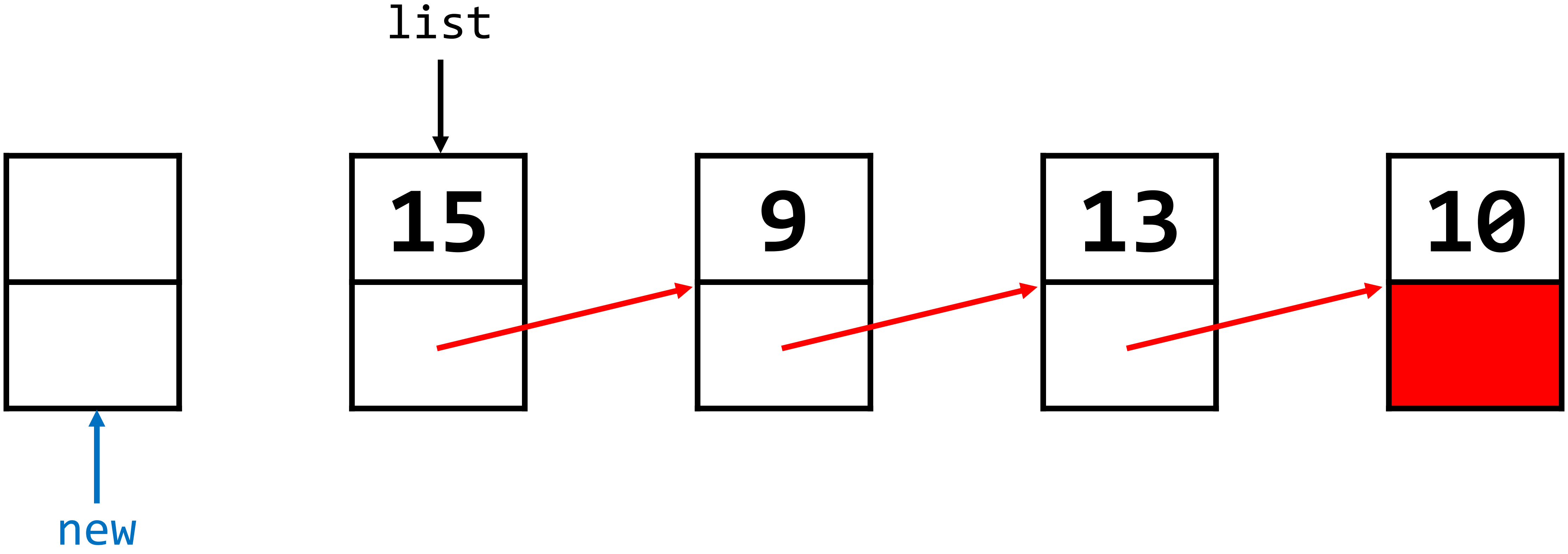
Singly-Linked Lists

```
list = insert(list, 12);
```



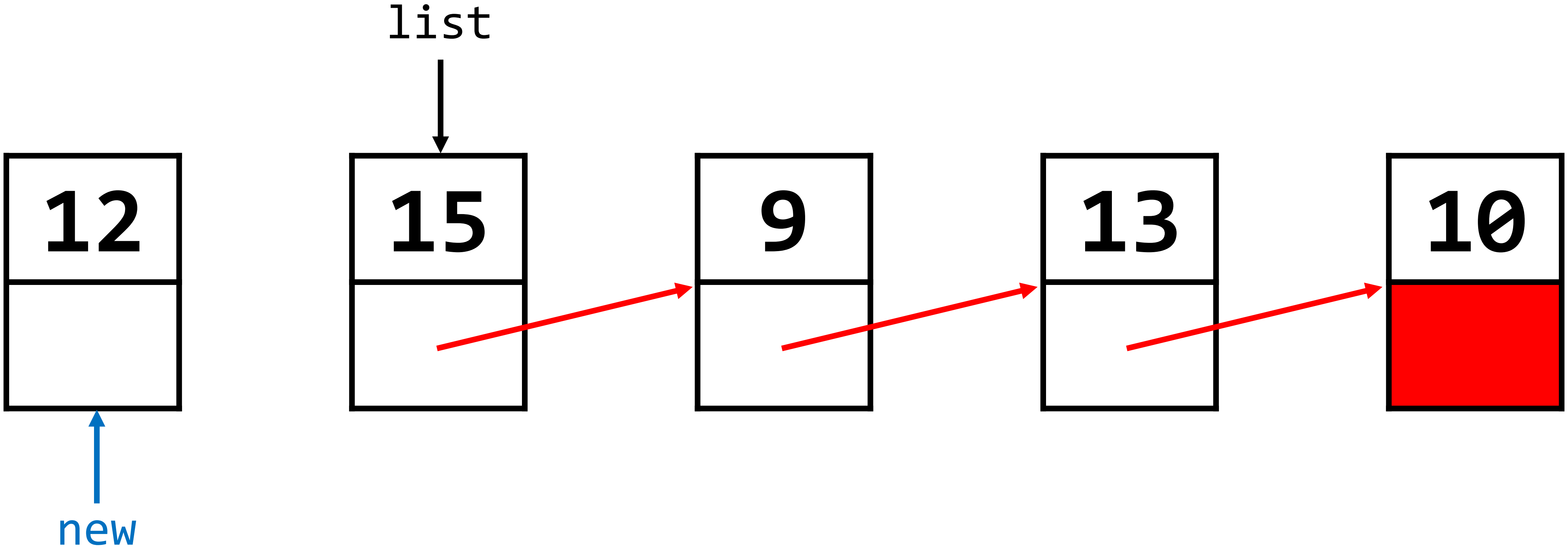
Singly-Linked Lists

```
list = insert(list, 12);
```



Singly-Linked Lists

```
list = insert(list, 12);
```

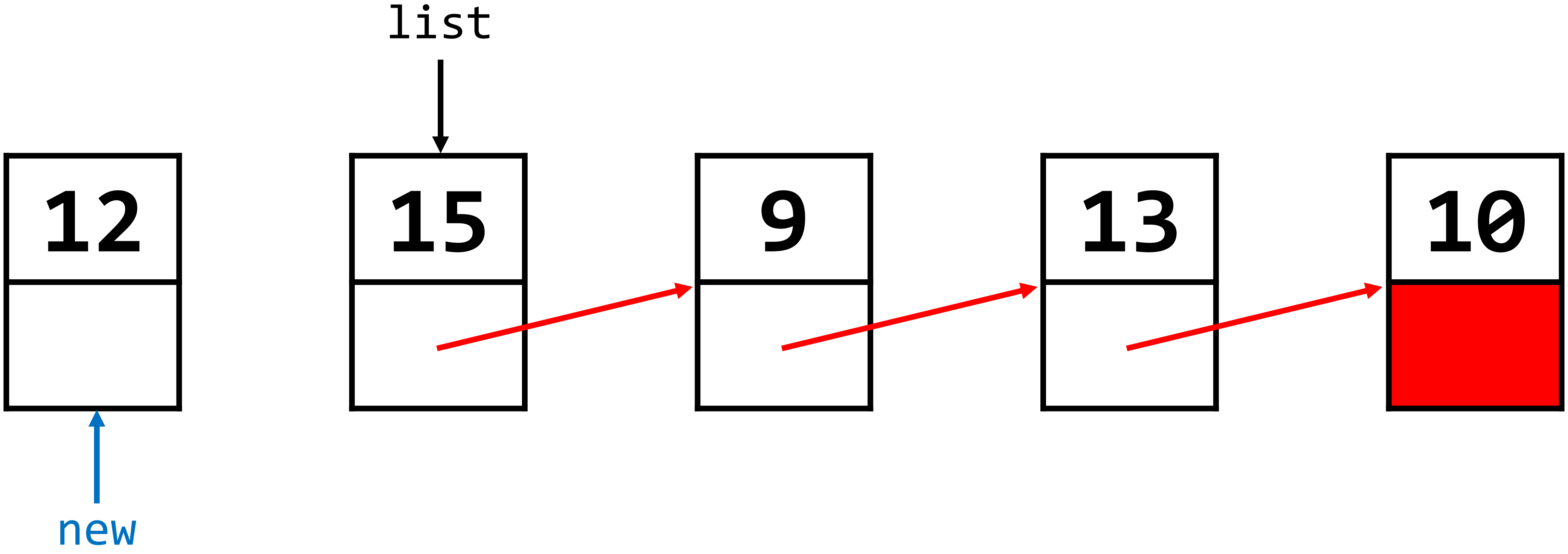


Singly-Linked Lists

- Decision time!
- Which pointer should we move first? Should the "12" node be the new head of the linked list, since it now exists, or should we connect it to the list first?
- This is one of the trickiest things with linked lists. Order matters!

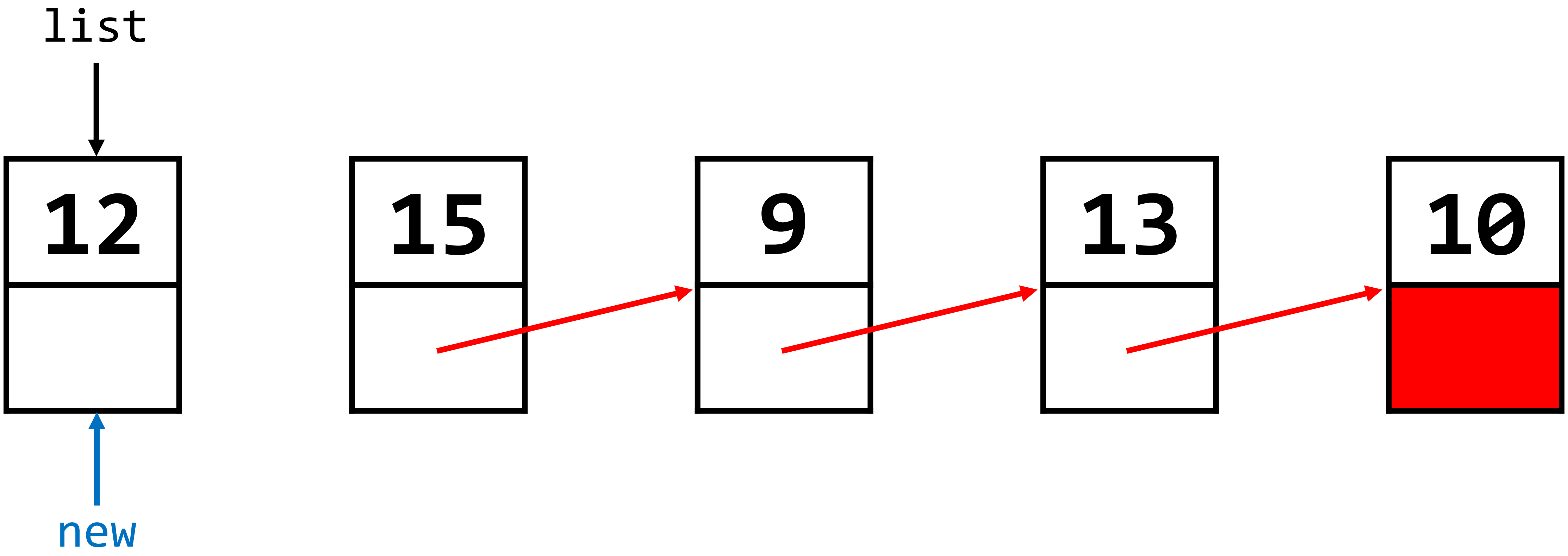
Singly-Linked Lists

`list = insert(list, 12);`



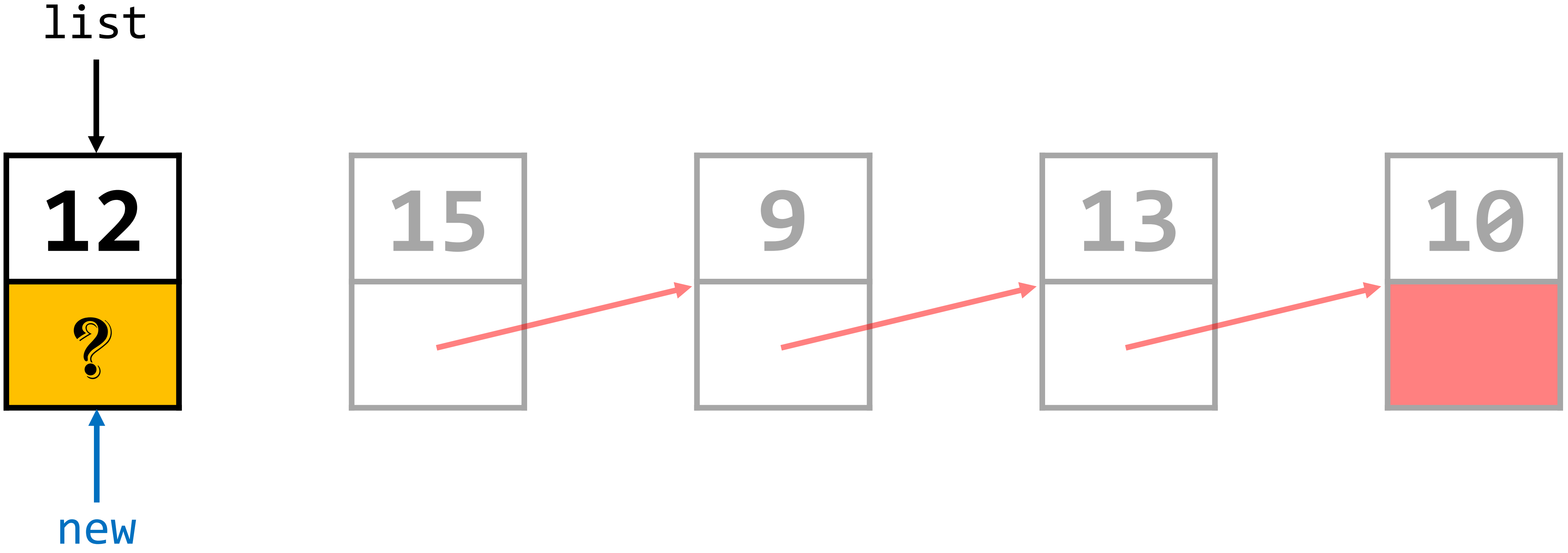
Singly-Linked Lists

`list = insert(list, 12);`



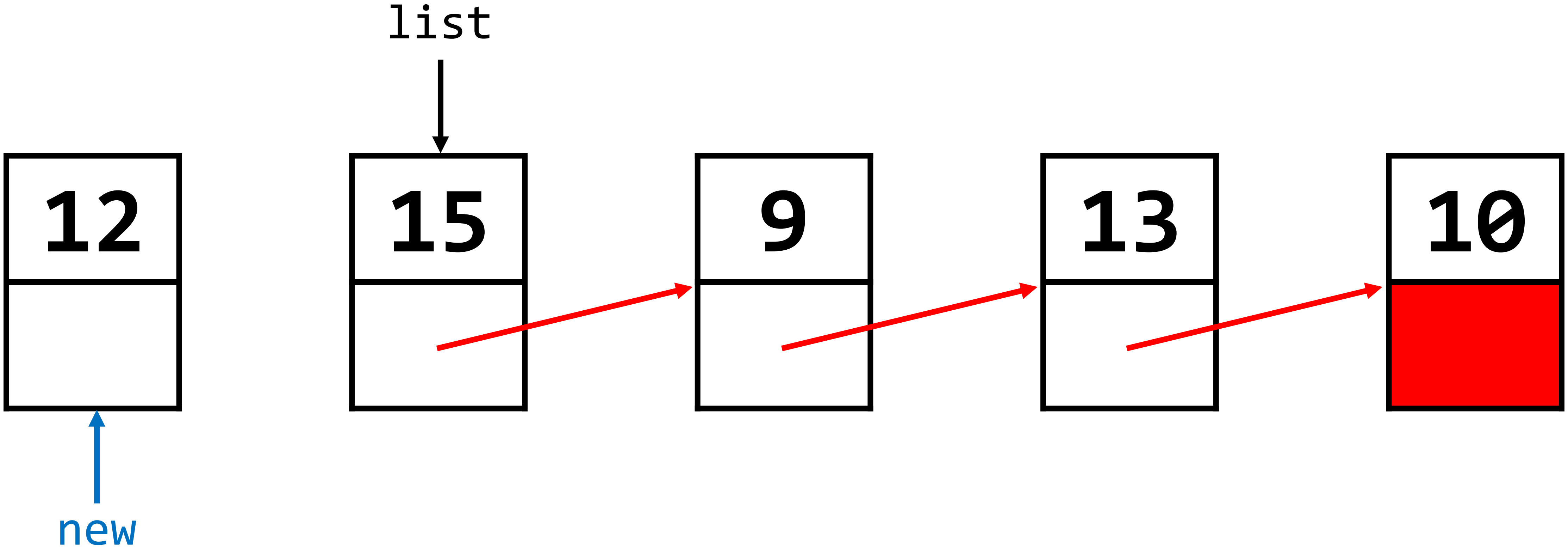
Singly-Linked Lists

```
list = insert(list, 12);
```



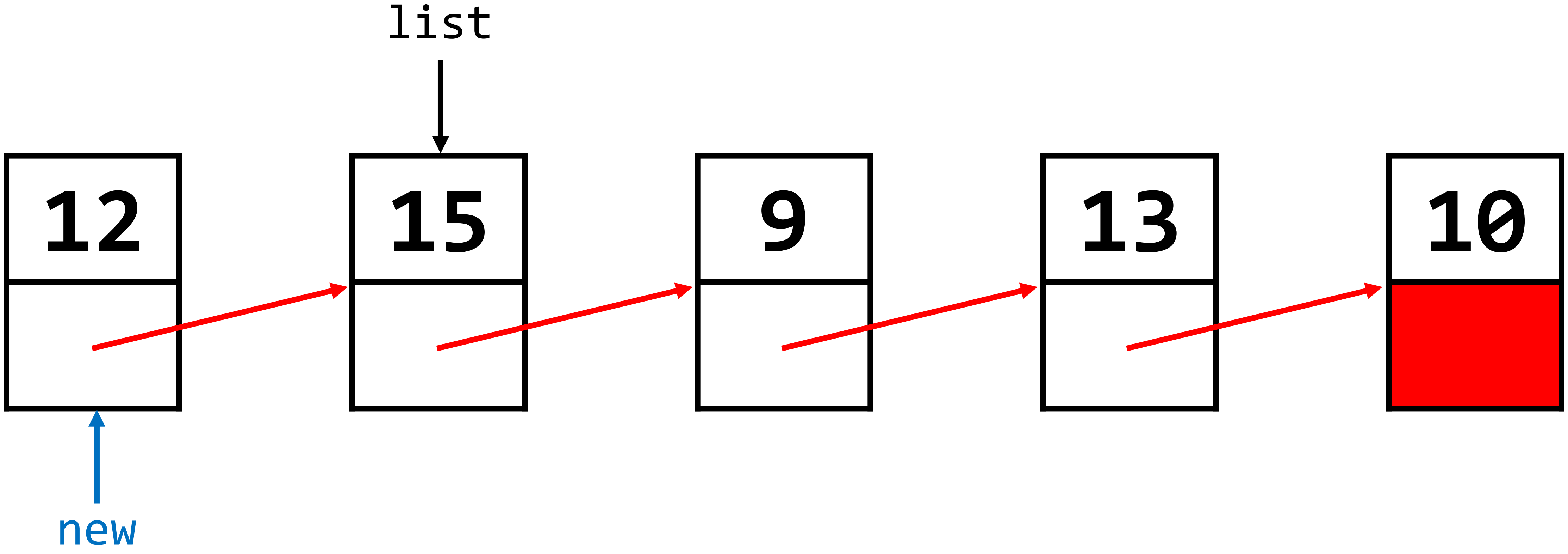
Singly-Linked Lists

```
list = insert(list, 12);
```



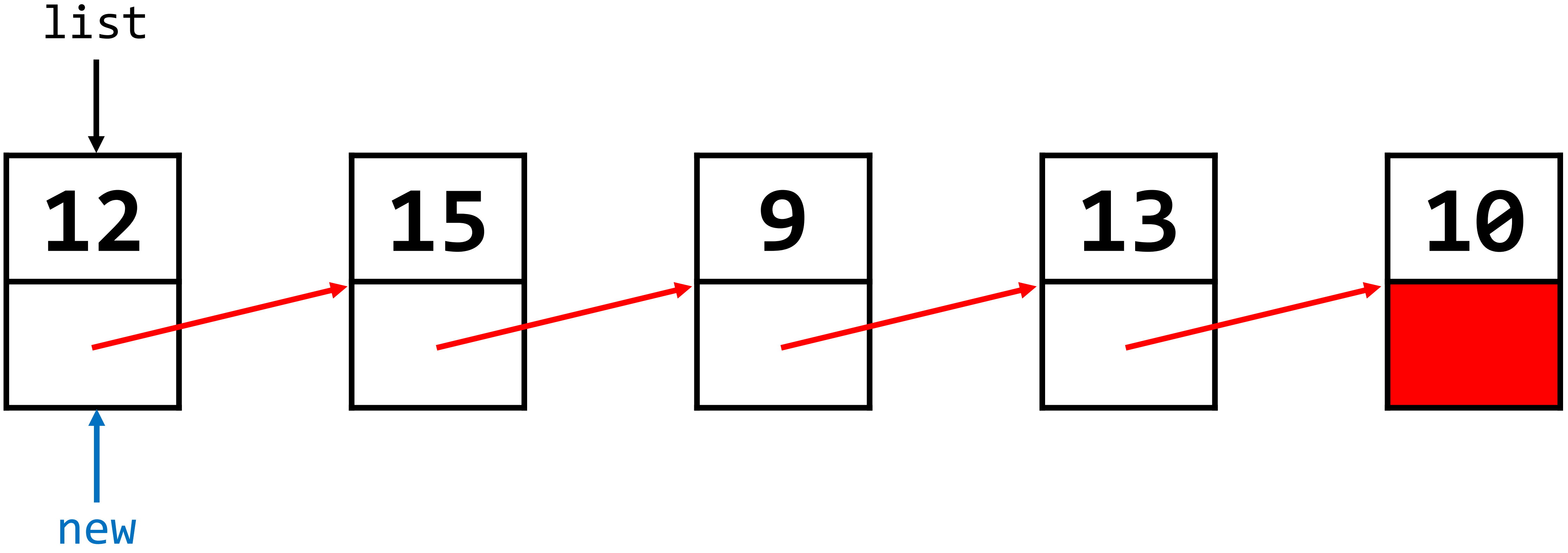
Singly-Linked Lists

```
list = insert(list, 12);
```



Singly-Linked Lists

```
list = insert(list, 12);
```



Singly-Linked Lists

- Delete an entire linked list.

```
void destroy(sllnode* head);
```

Singly-Linked Lists

- Delete an entire linked list.

```
void destroy(sllnode* head);
```

- Steps involved:
  a. If you’ve reached a null pointer, stop.
  b. Delete the rest of the list.
  c. Free the current node.

Singly-Linked Lists

- Delete an entire linked list.

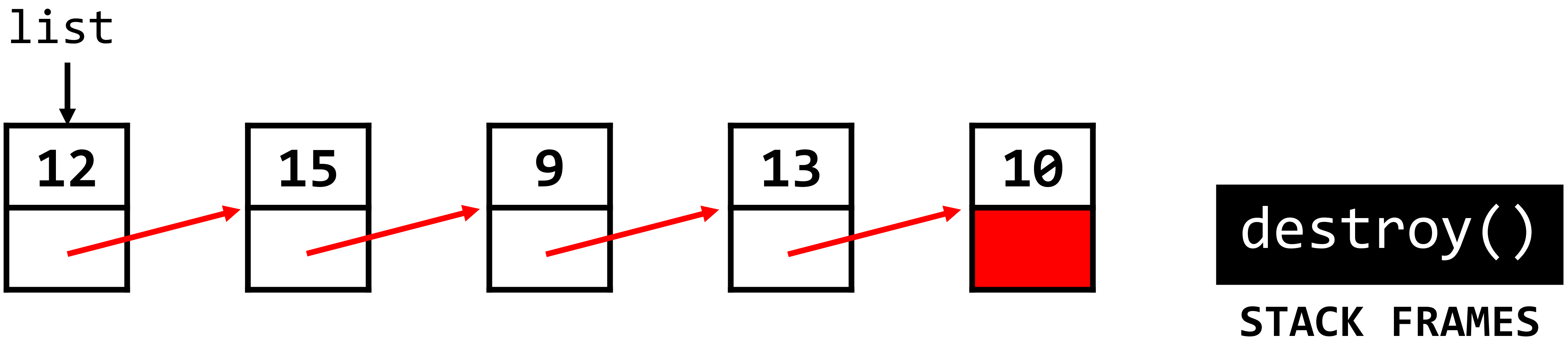
```
void destroy(sllnode* head);
```

- Steps involved:
  a. If you've reached a null pointer, stop.
  b. Delete the rest of the list.
  c. Free the current node.

Singly-Linked Lists

```
destroy(list);
```

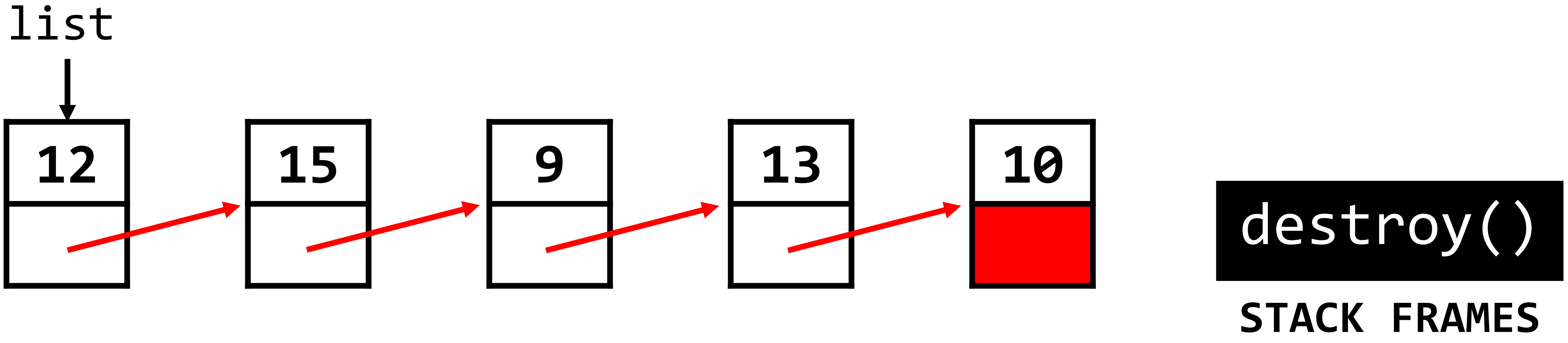
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

`destroy(list);`

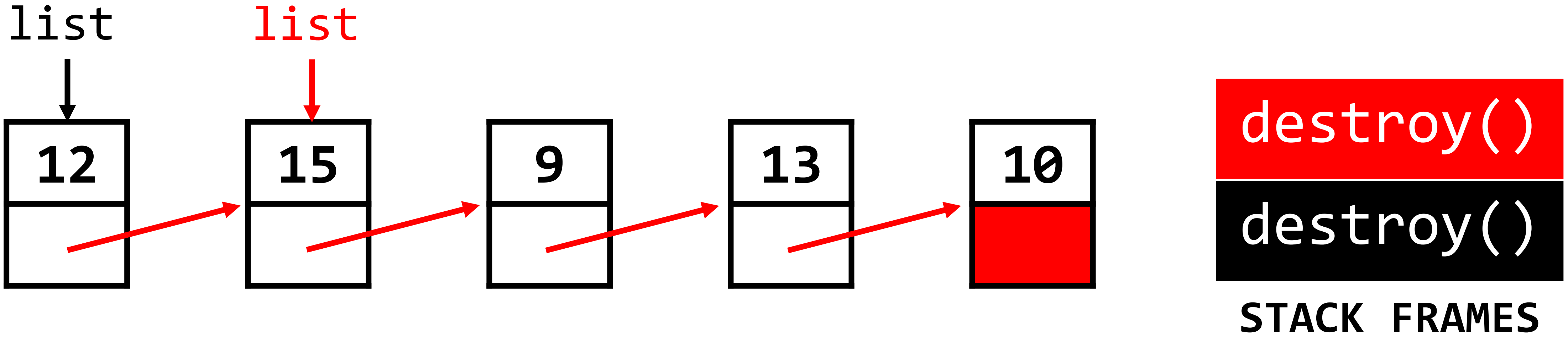
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

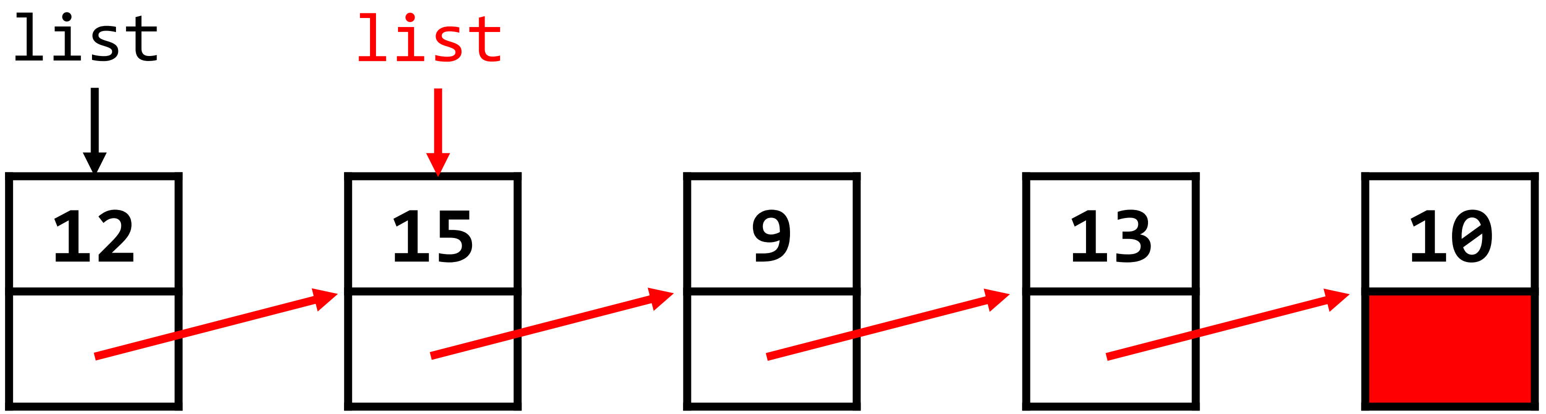
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

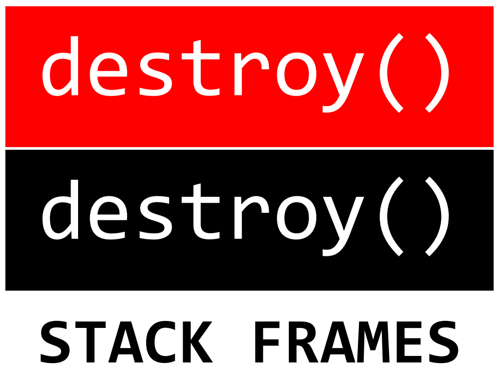


Singly-Linked Lists

```
destroy(list);
```

a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

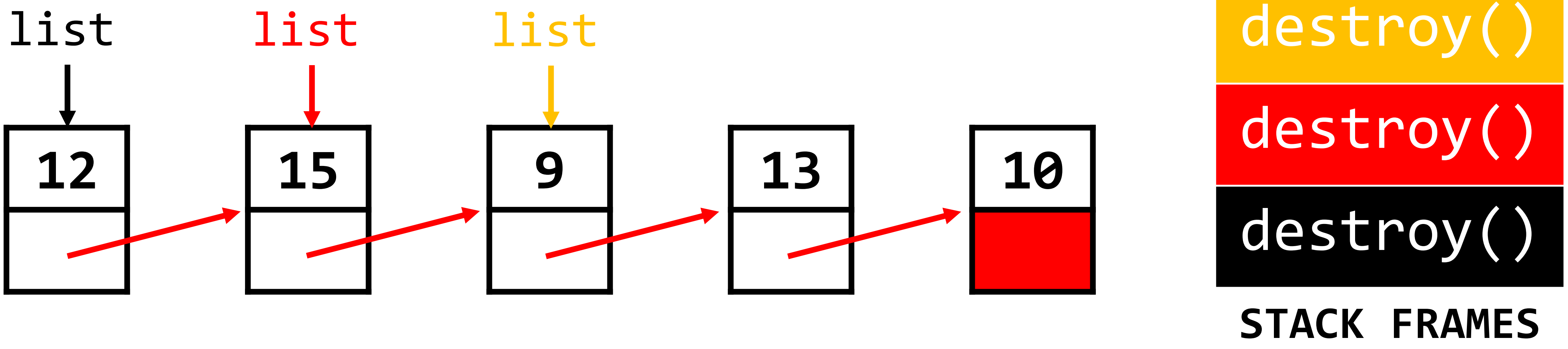




Singly-Linked Lists

```
destroy(list);
```

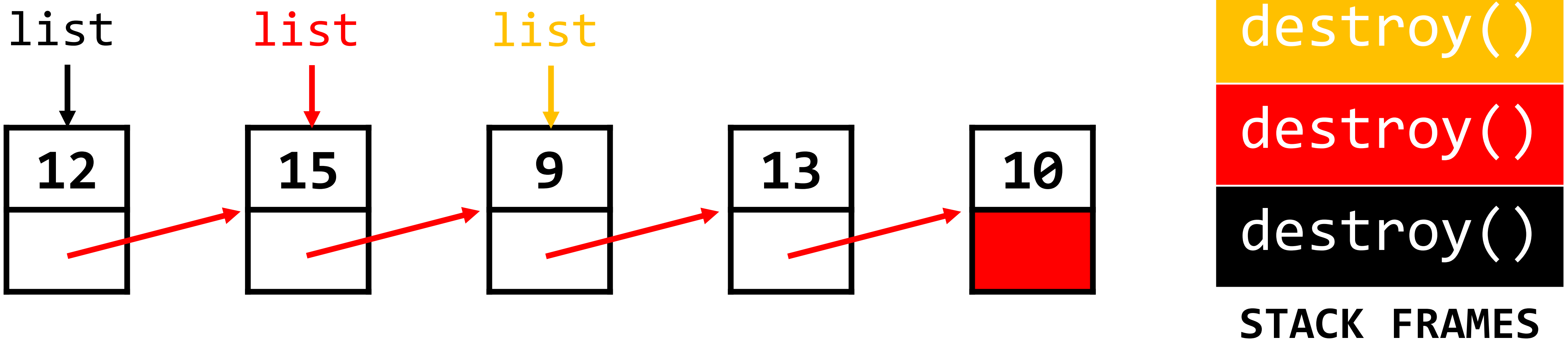
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

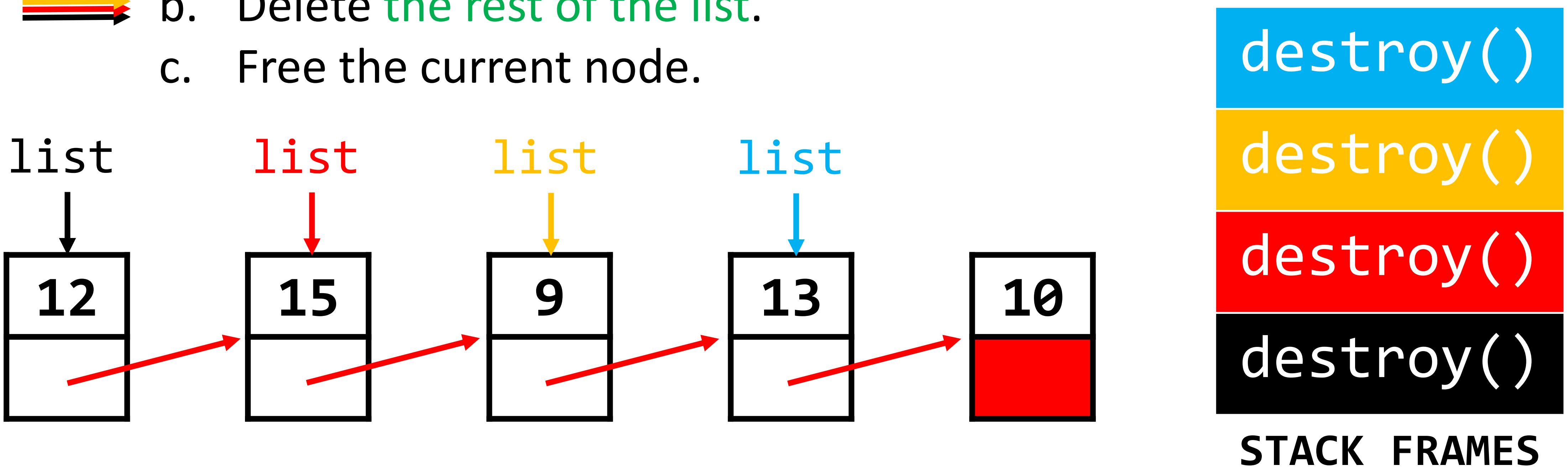
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

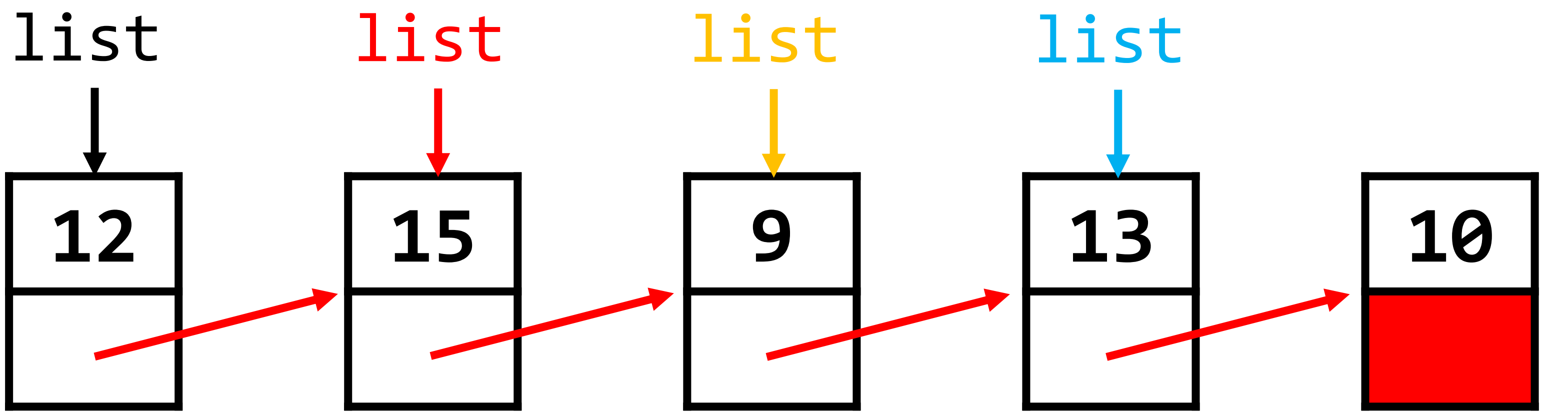
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

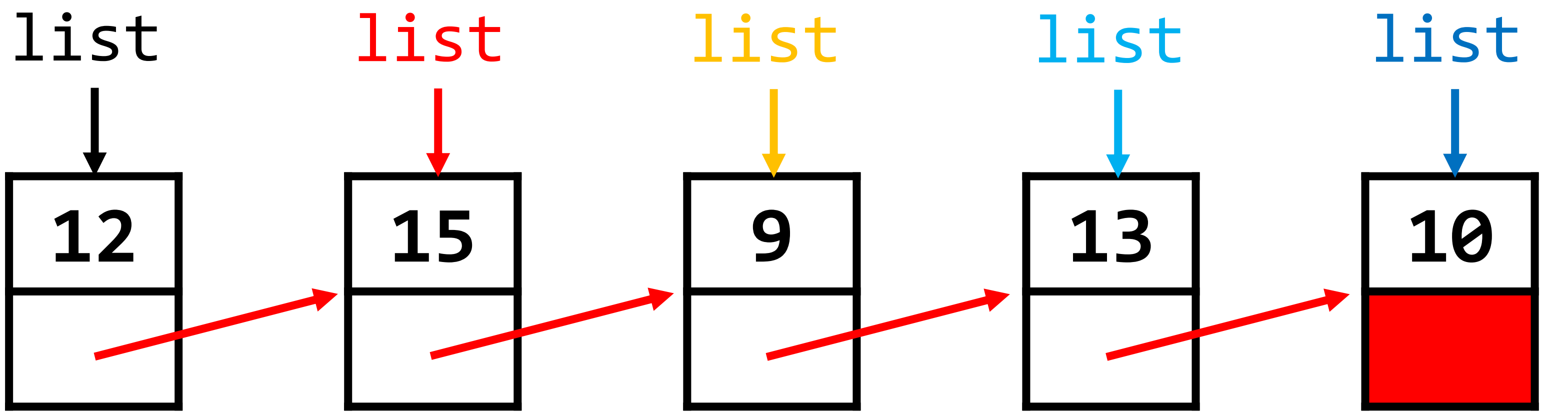
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

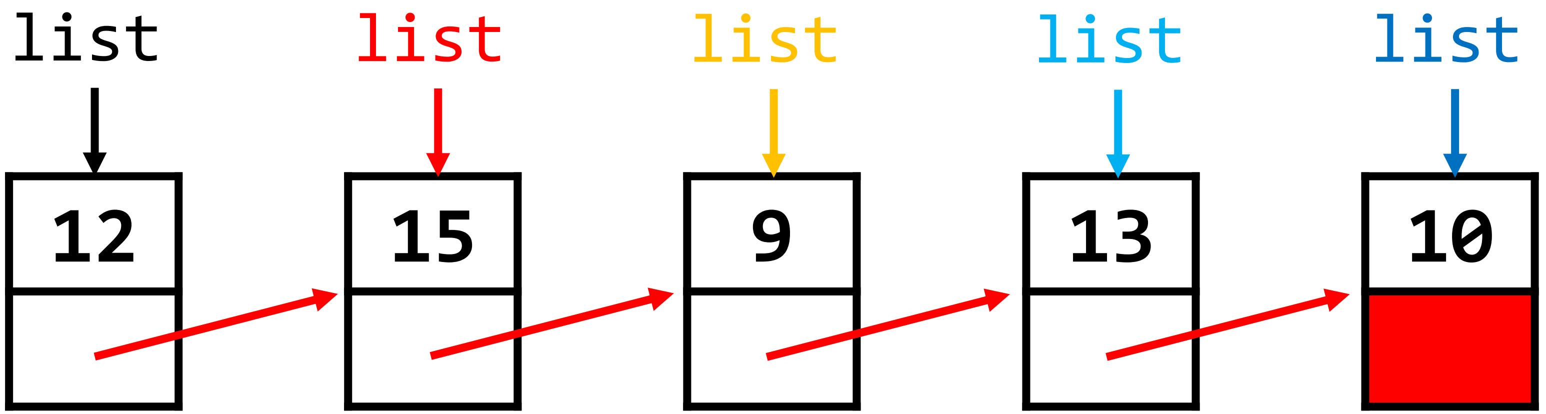
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

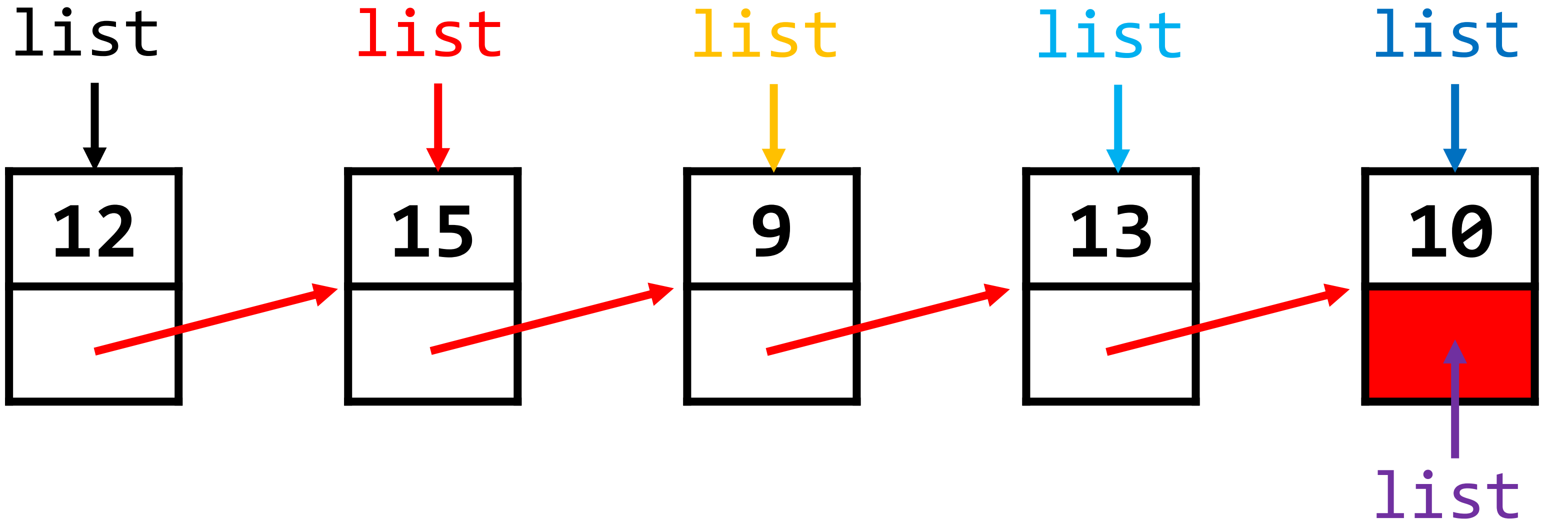
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

`destroy(list);`

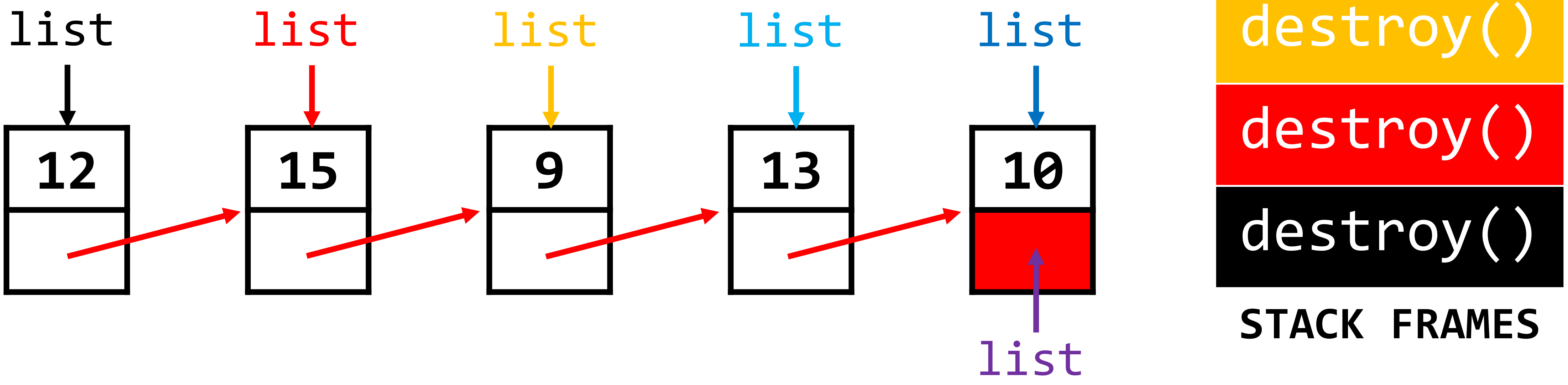
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

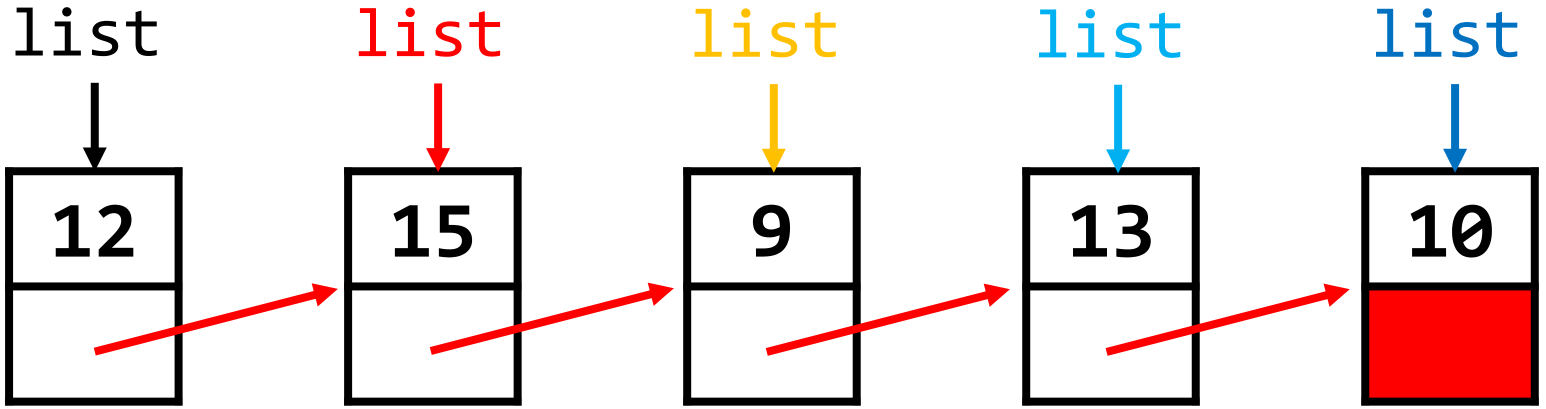
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

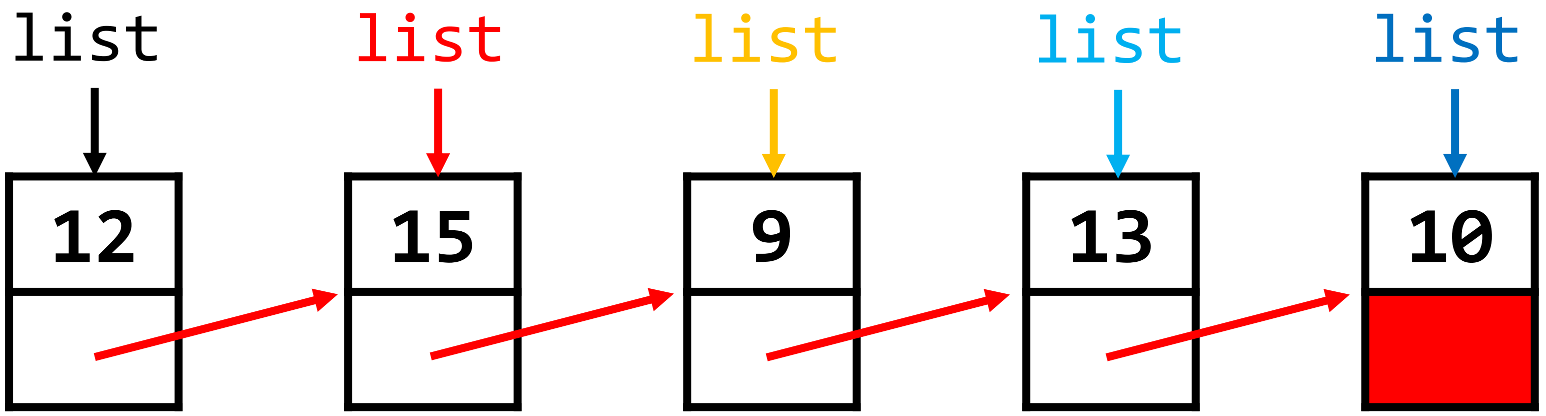
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

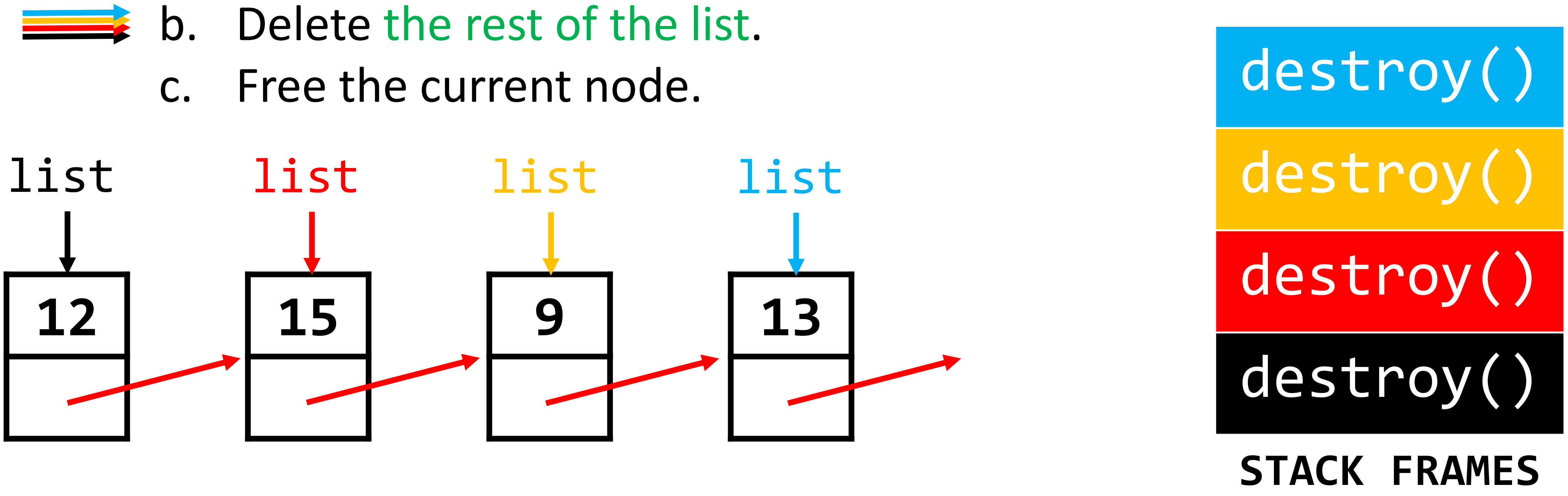
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

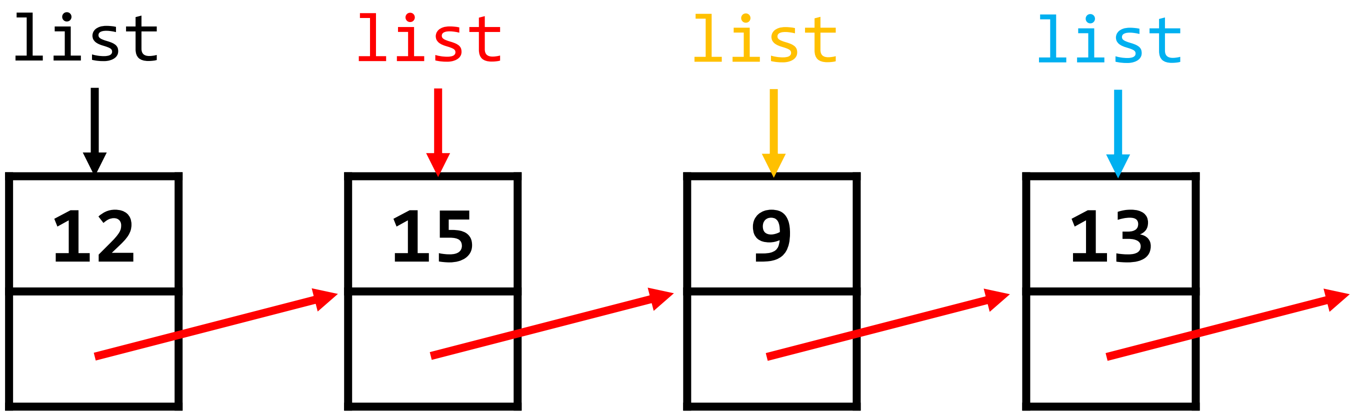
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

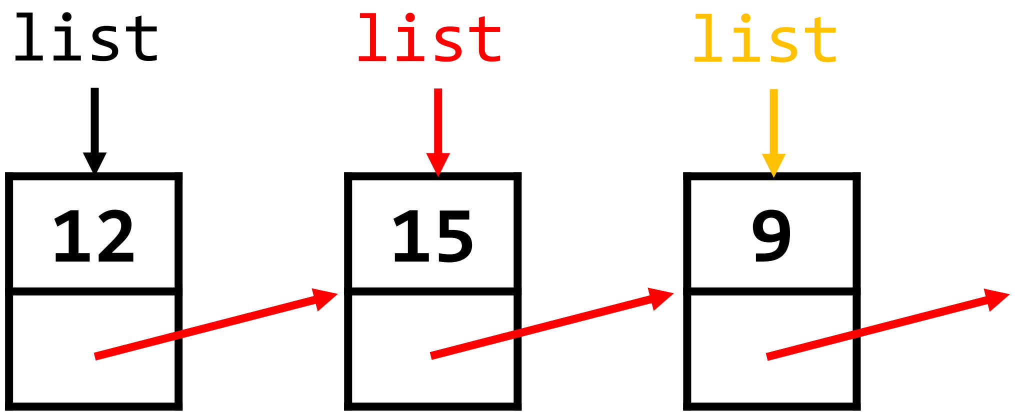
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

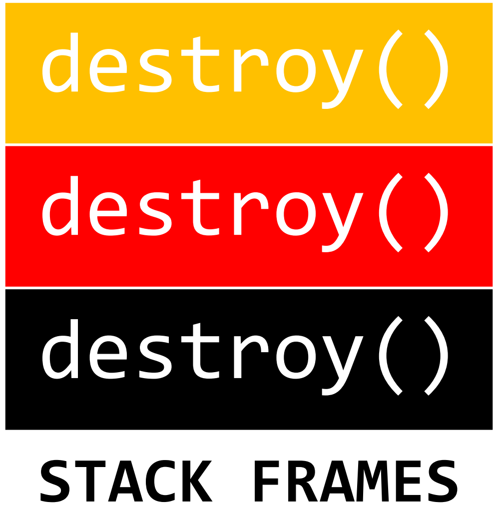


Singly-Linked Lists

```
destroy(list);
```

a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

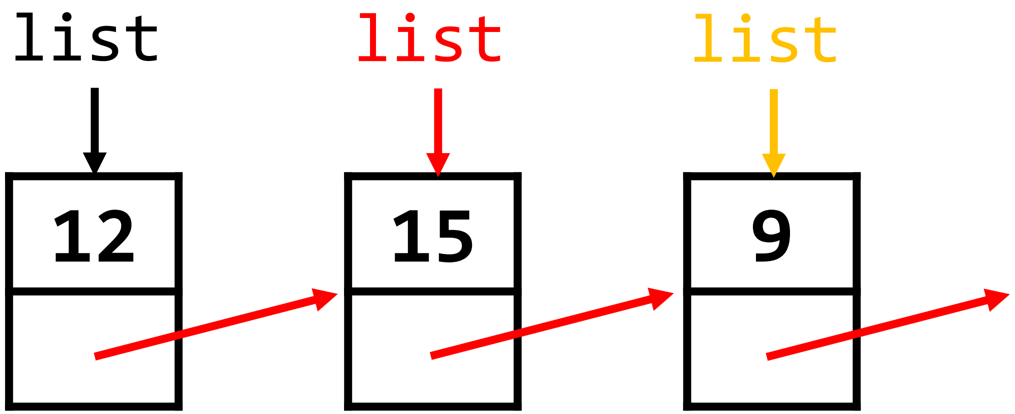


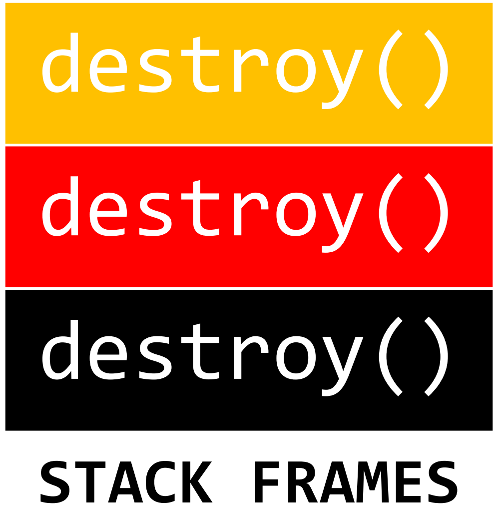


Singly-Linked Lists

```
destroy(list);
```

a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

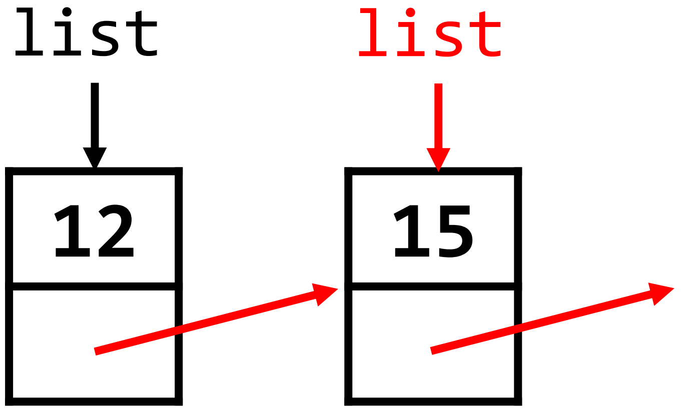


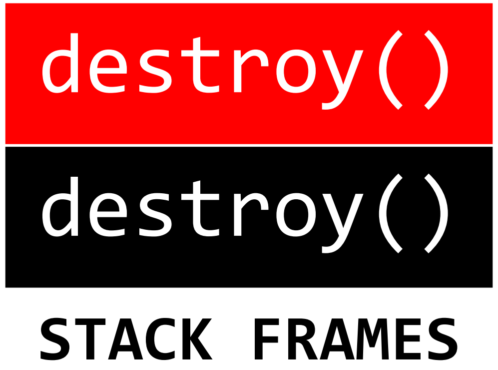


Singly-Linked Lists

```
destroy(list);
```

a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

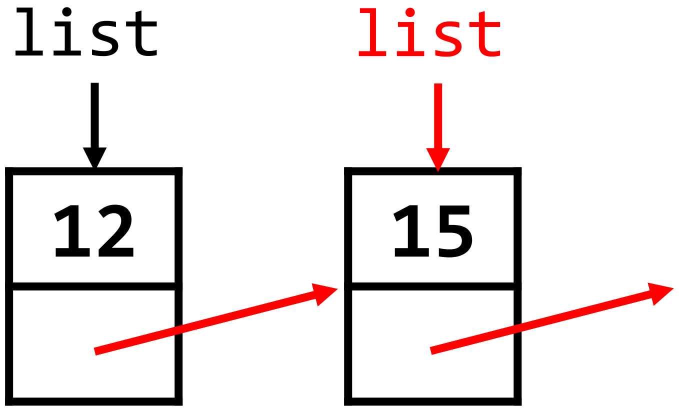


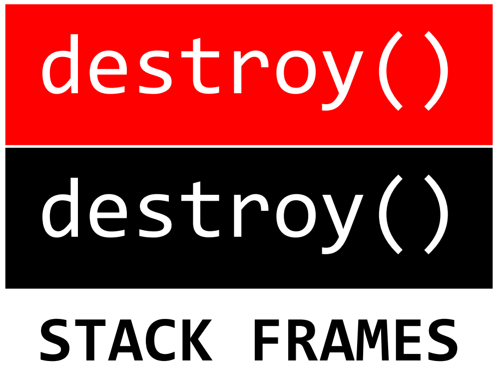


Singly-Linked Lists

```
destroy(list);
```

a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

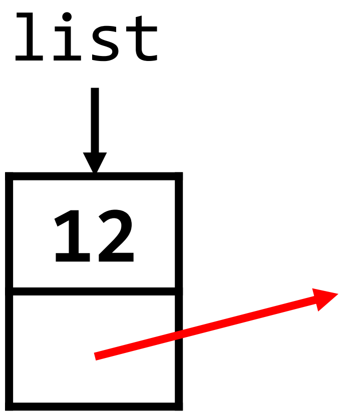




Singly-Linked Lists

```
destroy(list);
```

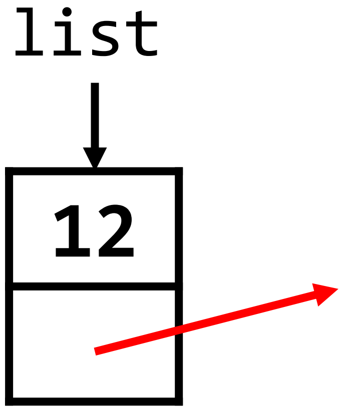
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

STACK FRAMES

Singly-Linked Lists

- In order to work with linked lists effectively, there are a number of operations that we need to understand:

1. Create a linked list when it doesn't already exist.
2. Search through a linked list to find an element.
3. Insert a new node into the linked list.
4. Delete a single element from a linked list.
5. Delete an entire linked list.