CSCI 2132: Software Development

Dynamic Memory Management

Norbert Zeh

Faculty of Computer Science
Dalhousie University

Winter 2019

The Heap

Memory region where we can freely request memory

`malloc`: Request a chunk of heap memory

`free`: Release a chunk of heap memory allocated using malloc
(Size information stored close to the allocated block)

Operating system keeps track of free (available) memory:

- **Simplest:** A linked list of free blocks (can be very slow)
- **Better:** Buddy system (CSCI 3136)

Pros and cons of heap allocation:

- **Pro:** very flexible, objects of arbitrary sizes, with arbitrary lifetimes
- **Con:** Heap management has a cost, can become the program's main bottleneck

Allocating and Freeing Memory

```
void *malloc(size_t num_bytes);
```

- **Argument:** N
- **Return value** if out of memory

```
void free(voi
```

- **Argument:** P
  (Must have been alloc ed using malloc)

```
#include <stdlib.h>

int main() {
  int *array = malloc(1000 * sizeof(int));
  for (int i = 0; i < 1000; ++i)
    array[i] = i;
  free(array);
  return 0;
}
```

malloc returns a `void *`.

You assign to an `int *`.

Some compilers may require you to include an explicit type cast `(int *)` here.

You need to figure out how many bytes you need.

Don’t forget to free the memory.

More Allocation Functions

`void *calloc(size_t num_elems, size_t elem_size)`:

- Allocates space for an array of objects
- Sets all allocated bytes to 0

`void *realloc(void *ptr, size_t size)`:

- “Resizes” the block referenced by ptr to size
- Growing and shrinking is allowed
- The location of the block may change!
  (Use `ptr = realloc(ptr, size)`)
- `ptr == NULL` ⇒ `realloc` behaves like `malloc`
- `size == 0` ⇒ `realloc` behaves like `free`

Resizable Arrays

Vectors in C++, Java, Scala, Rust, … grow automatically to accommodate more items.

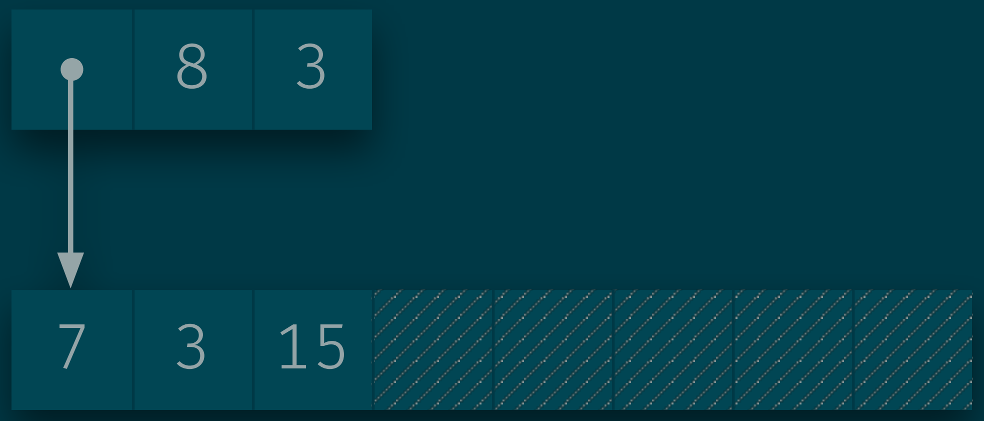
C arrays do not support this.

How are these resizable vectors implemented?

Supported operations

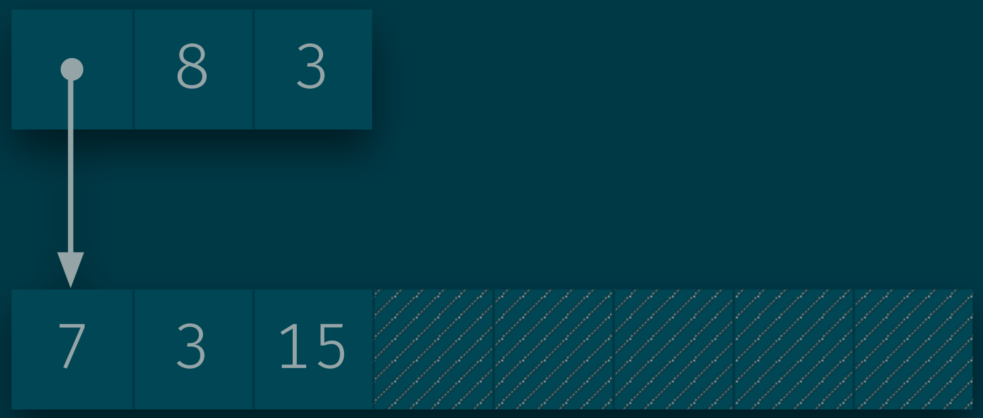
`push(array, item)`	Add a new item to the end of the array
`pop(array)`	Remove the last item from the array
`get(array, index)`	Retrieve the item at the given index
`put(array, index, item)`	Update the item at the given index

The Data Structure



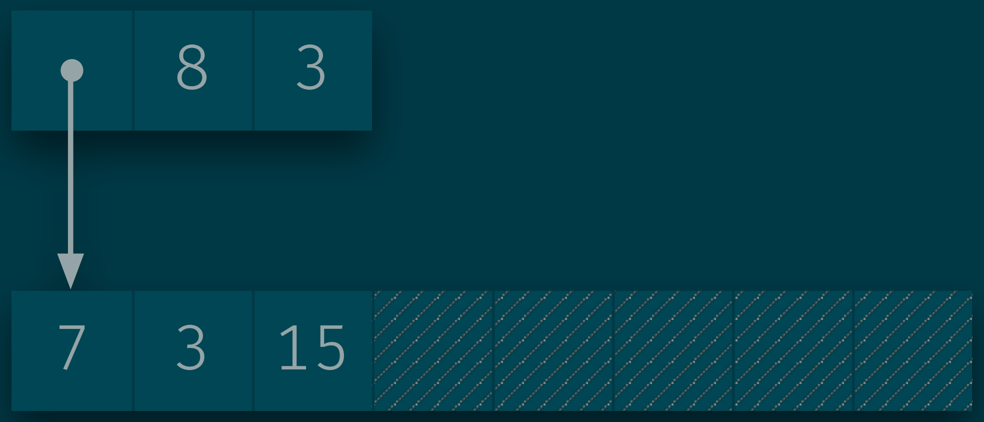
```
typedef struct _vec_t *vec_t;
struct _vec_t {
    int *data;
    size_t capacity, size;
};
```

Creating and Destroying a Vector



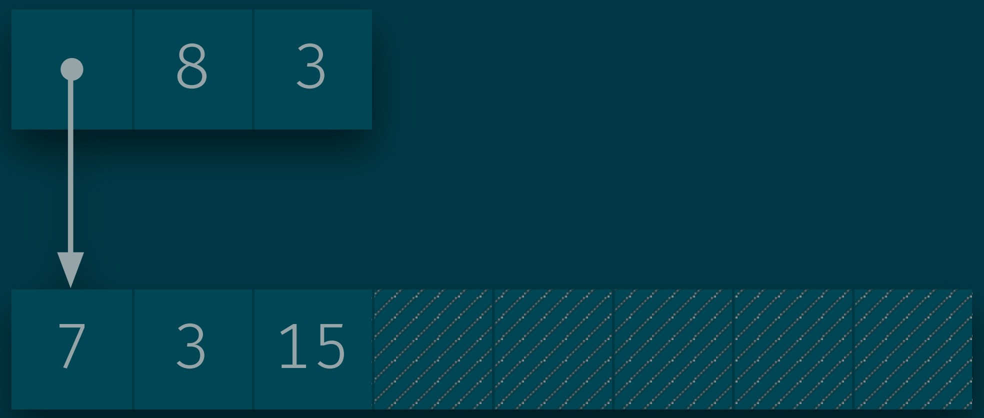
```
vec_t make_vector() {
    vec_t vec = malloc(sizeof(struct _vec_t));
    vec->data = malloc(8 * sizeof(int));
    vec->capacity = 8; vec->size = 0;
    return vec;
}

void destroy_vector(vec_t vec) {
    free(vec->data); free(vec);
}
```

Get and Put



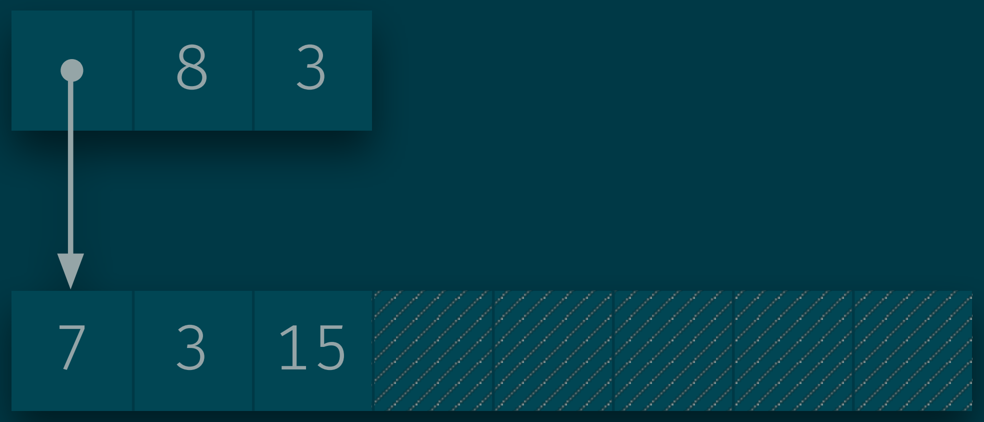
```
int get(vec_t vec, unsigned int index) {
    return vec->data[index];
}

void put(vec_t vec, unsigned int index, int val) {
    vec->data[index] = val;
}
```

Push



```c
void push(vec_t vec, int item) {
    if (vec->size == vec->capacity) {
        vec->capacity *= 2;
        vec->data = realloc(
            vec->data,
            vec->capacity * sizeof(int));
    }
    vec->data[vec->size++] = item;
}
```

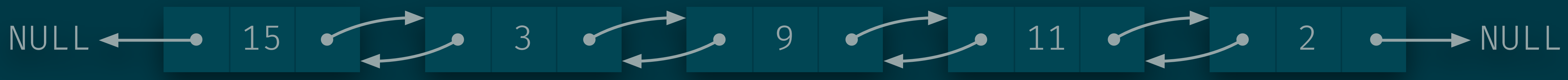
Pop



```
void pop(vec_t vec) {
    --vec->size;
    if (vec->size <= vec->capacity / 4 &&
            vec->capacity > 8) {
        vec->capacity /= 2;
        vec->data = realloc(
            vec->data,
            vec->capacity * sizeof(int));
    }
}
```

A Doubly-Linked List

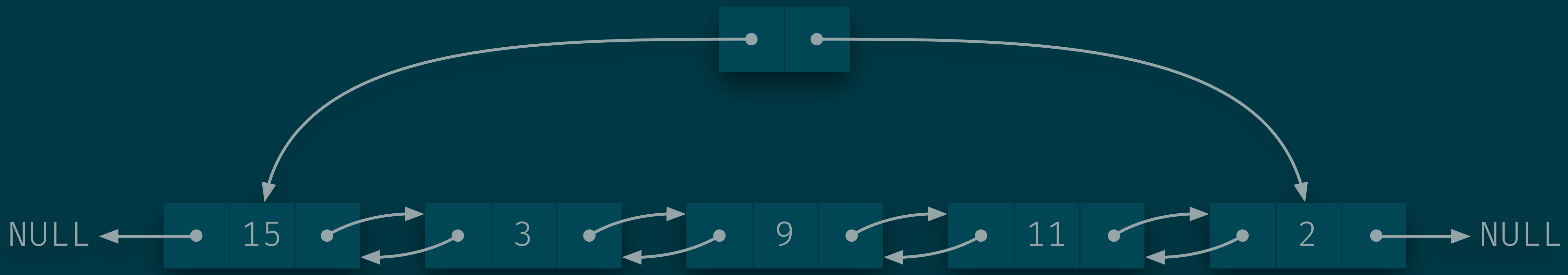
A doubly-linked list stores a sequence of items



Supported operations

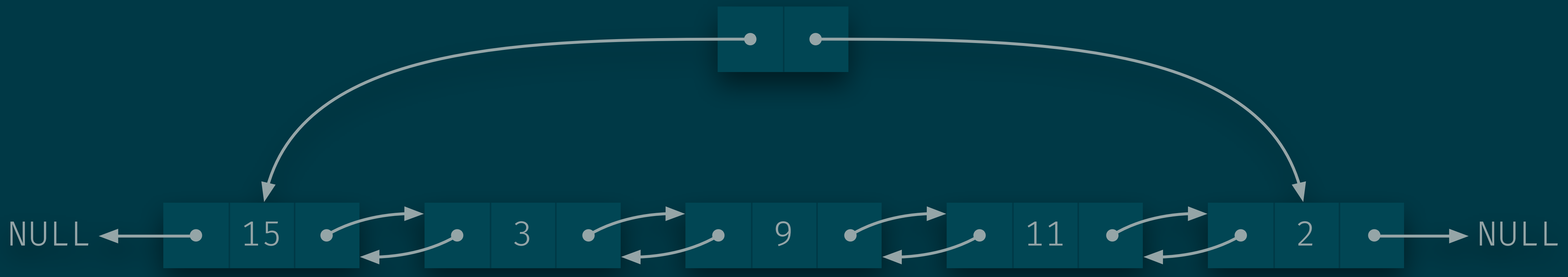
`append(list, item)`	Add item at the end of the list
`prepend(list, item)`	Add item at the start of the list
`insert_after(list, node, item)`	Add item after the given node
`delete(list, node)`	Delete the given node
`head(list)`	Access the first node of the list
`tail(list)`	Access the last node of the list
`get_item(node)`	Get the item stored at a node
`pred(node)`	Get the node before this node
`succ(node)`	Get the node after this node

The List and its Nodes



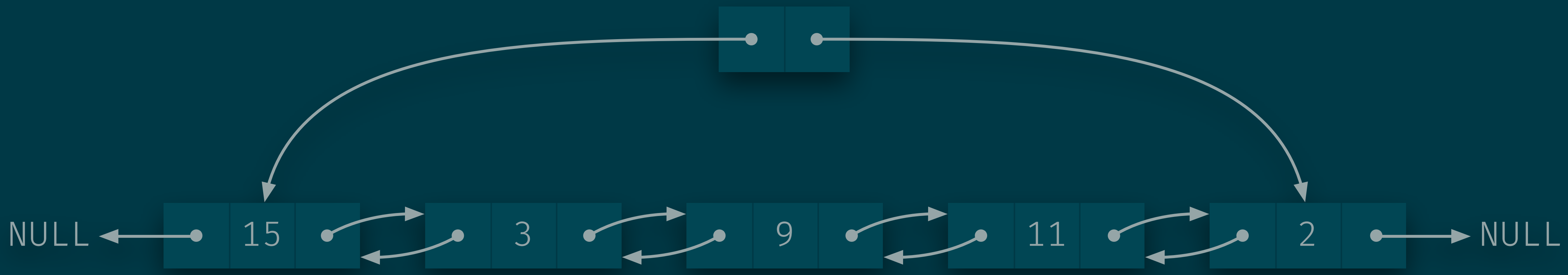
```
typedef struct _node_t *node_t;
struct _node_t {
    int val;
    node_t pred, succ;
};

typedef struct _list_t *list_t;
struct _list_t {
    node_t head, tail;
};
```

Creating and Destroying a List



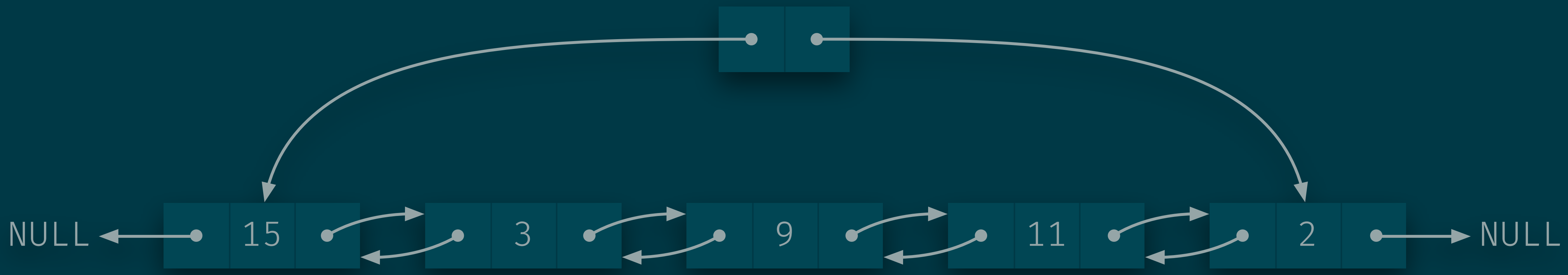
```
list_t make_list() {
    list_t list = malloc(sizeof(struct _list_t));
    list->head = list->tail = NULL;
    return list;
}

void destroy_list(list_t list) {
    node_t curr, next;
    for (curr = list->head; curr != null; curr = next) {
         next = curr->succ; free(curr); }
    free(list);
}
```

Append Operation



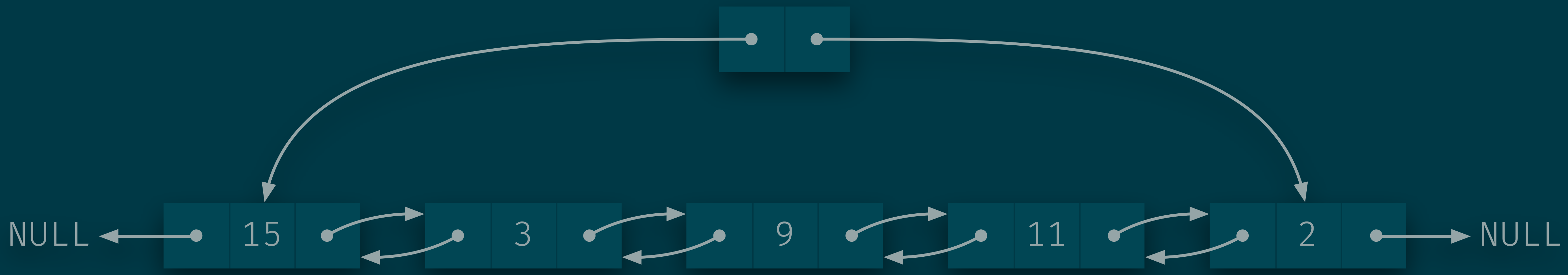
```
node_t append(list_t list, int val) {
    node_t new_node = malloc(sizeof(struct _node_t));
    new_node->val  = val;
    new_node->succ = NULL;
    new_node->pred = list->tail;
    if (list->tail)
        list->tail->succ = new_node;
    else
        list->head = new_node;
    list->tail = new_node;
    return new_node;
}
```

Insert Operation



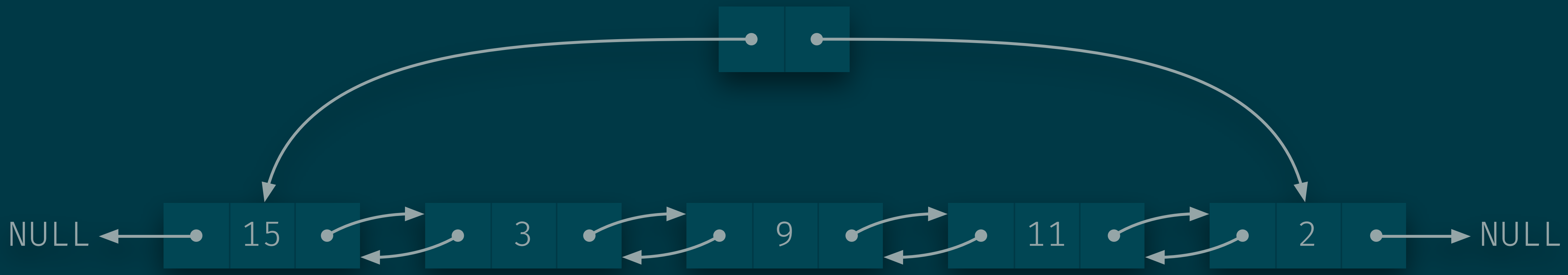
```
node_t insert_after(list_t list, node_t node, int val) {
    node_t new_node = malloc(sizeof(struct _node_t));
    new_node->val  = val;
    new_node->succ = node->succ;
    new_node->pred = node;
    node->succ = new_node;
    if (list->tail == node)
        list->tail = new_node;
    else
        new_node->succ->pred = new_node;
    return new_node;
}
```

Delete Operation



```
void delete(list_t list, node_t node) {
    if (node == list->head)
        list->head = node->succ;
    else
        node->pred->succ = node->succ;
    if (node == list->tail)
        list->tail = node->pred;
    else
        node->succ->pred = node->pred;
    free(node);
}
```

The Other Operations



Supported operations

`append(list, item)`	Add item at the end of the list
`prepend(list, item)`	Add item at the start of the list
`insert_after(list, node, item)`	Add item after the given node
`delete(list, node)`	Delete the given node
`head(list)`	Access the first node of the list
`tail(list)`	Access the last node of the list
`get_item(node)`	Get the item stored at a node
`pred(node)`	Get the node before this node
`succ(node)`	Get the node after this node