CSC 212: Data Structures and Abstractions

Linked Lists

Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2020

```python
import time

n = 100000


start = time.time()
array = []
for i in range(n):
    array.append('s')
print(time.time() - start)


start = time.time()
array = []
for i in range(n):
    array = array + ['s']
print(time.time() - start)
```

How are lists implemented in CPython?

CPython's lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure.

This makes indexing a list `a[i]` an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don't require an actual resize.

CPython is the reference implementation of the Python programming language

https://docs.python.org/3.8/faq/design.html#how-are-lists-implemented-in-cpython

Some STL Containers …

std::array

Defined in header <array>

```
template<
    class T,
    std::size_t N
> struct array;
```
(since C++11)

std::array is a container that encapsulates fixed size arrays.

```cpp
#include <string>
#include <iterator>
#include <iostream>
#include <algorithm>
#include <array>

int main()
{
    // construction uses aggregate initialization
    std::array<int, 3> a1{ {1, 2, 3} }; // double-braces required in C++11 prior to the CWG 1270 revision
                                        // (not needed in C++11 after the revision and in C++14 and beyond)
    std::array<int, 3> a2 = {1, 2, 3};  // never required after =
    std::array<std::string, 2> a3 = { std::string("a"), "b" };

    // container operations are supported
    std::sort(a1.begin(), a1.end());
    std::reverse_copy(a2.begin(), a2.end(), std::ostream_iterator<int>(std::cout, " "));

    std::cout << '\n';

    // ranged for loop is supported
    for(const auto& s: a3)
        std::cout << s << ' ';
}
```

https://en.cppreference.com/w/cpp/container/array

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```
(1)

```
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++17)

1) `std::vector` is a sequence container that encapsulates dynamic size arrays.

2) `std::pmr::vector` is an alias template that uses a polymorphic allocator

```cpp
#include <iostream>
#include <vector>

int main()
{
    // Create a vector containing integers
    std::vector<int> v = {7, 5, 16, 8};

    // Add two more integers to vector
    v.push_back(25);
    v.push_back(13);

    // Iterate and print values of vector
    for(int n : v) {
        std::cout << n << '\n';
    }
}
```

std::forward_list

Defined in header <forward_list>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class forward_list;
```
(1) (since C++11)

```
namespace pmr {
    template <class T>
    using forward_list = std::forward_list<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++17)

`std::forward_list` is a container that supports fast insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is implemented as a singly-linked list and essentially does not have any overhead compared to its implementation in C. Compared to `std::list` this container provides more space efficient storage when bidirectional iteration is not needed.

```cpp
#include <forward_list>
#include <iostream>

int main() {
    std::forward_list<int> numbers;
    std::cout << "Initially, numbers.empty(): " << numbers.empty() << '\n';
    numbers.push_front(42);
    numbers.push_front(13317);
    std::cout << "After adding elements, numbers.empty(): " << numbers.empty() << '\n';
}
```

std::list

Defined in header <list>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class list;
```
(1)

```
namespace pmr {
    template <class T>
    using list = std::list<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++17)

`std::list` is a container that supports constant time insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is usually implemented as a doubly-linked list. Compared to `std::forward_list` this container provides bidirectional iteration capability while being less space efficient.

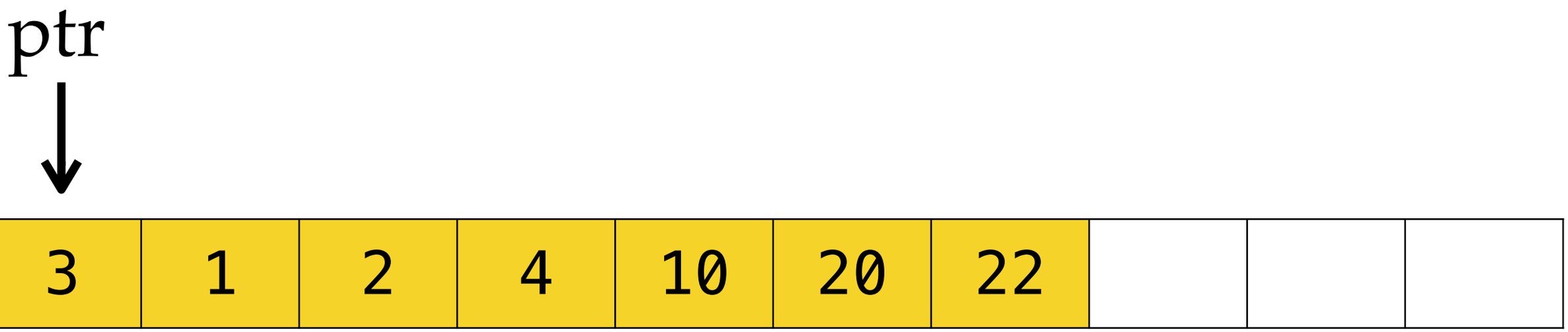
```
#include <algorithm>
#include <iostream>
#include <list>

int main() {
    // Create a list containing integers
    std::list<int> l = { 7, 5, 16, 8 };
    // Add an integer to the front of the list
    l.push_front(25);
    // Add an integer to the back of the list
    l.push_back(13);
    // Insert an integer before 16 by searching
    auto it = std::find(l.begin(), l.end(), 16);
    if (it != l.end()) {
        l.insert(it, 42);
    }
    // Iterate and print values of the list
    for (int n : l) {
        std::cout << n << '\n';
    }
}
```

Linked Lists

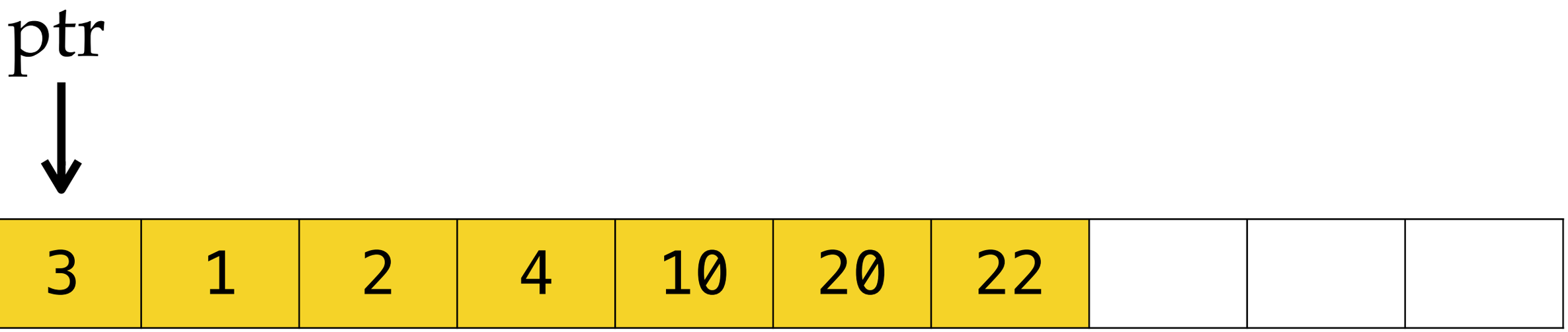
Arrays

- Think about making **insertions** efficiently, what is the computational cost of inserting 1 element?
  - rear?
  - **front?**
  - **middle?**



Arrays

- Think about making **deletions** efficiently, what is the computational cost of deleting 1 element?
  - rear?
  - **front?**
  - **middle?**



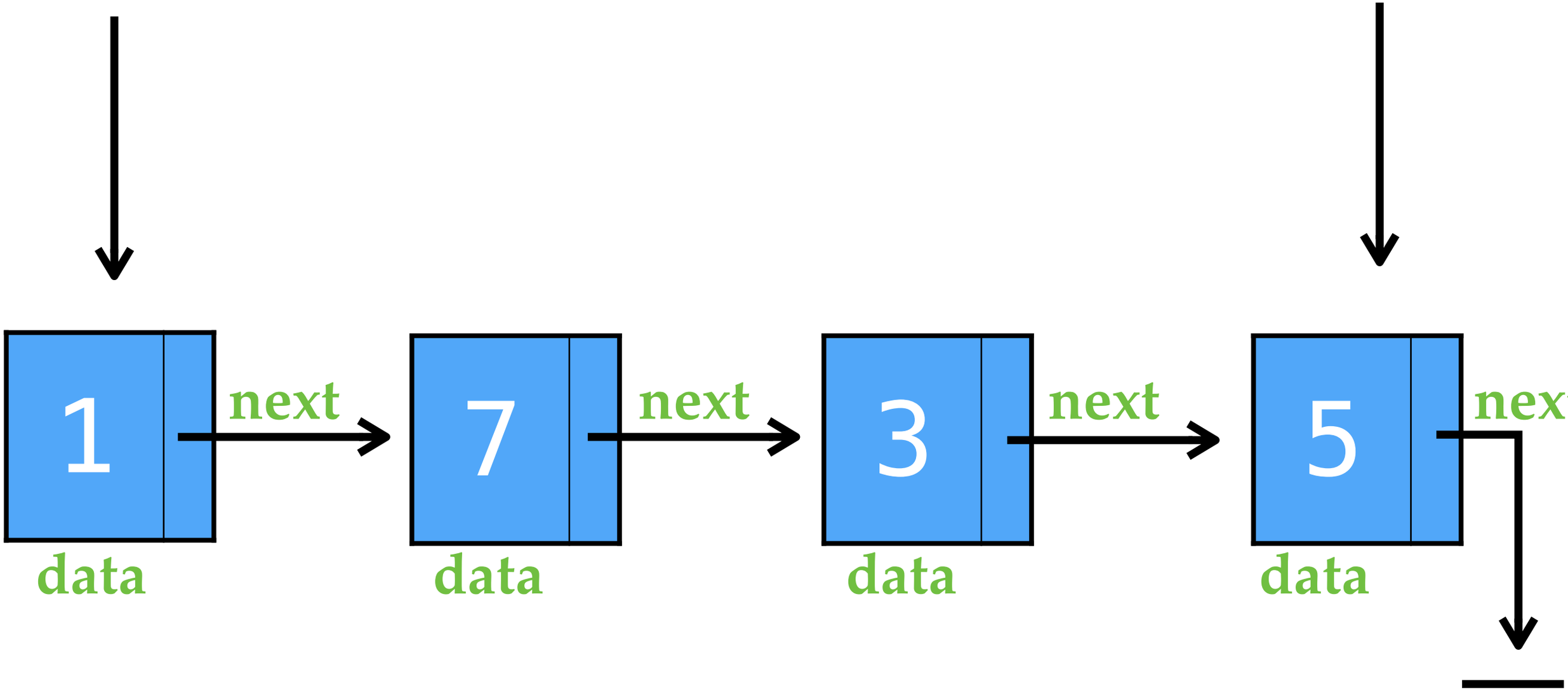
Linked Lists

- Collections of sequential elements stored at **non-contiguous** locations in memory
- Elements are stored in **nodes**
- Nodes are connected by **links**
  - every node keeps a pointer to the next node
- Can **grow** and **shrink** dynamically
- Allow for fast insertions/deletions

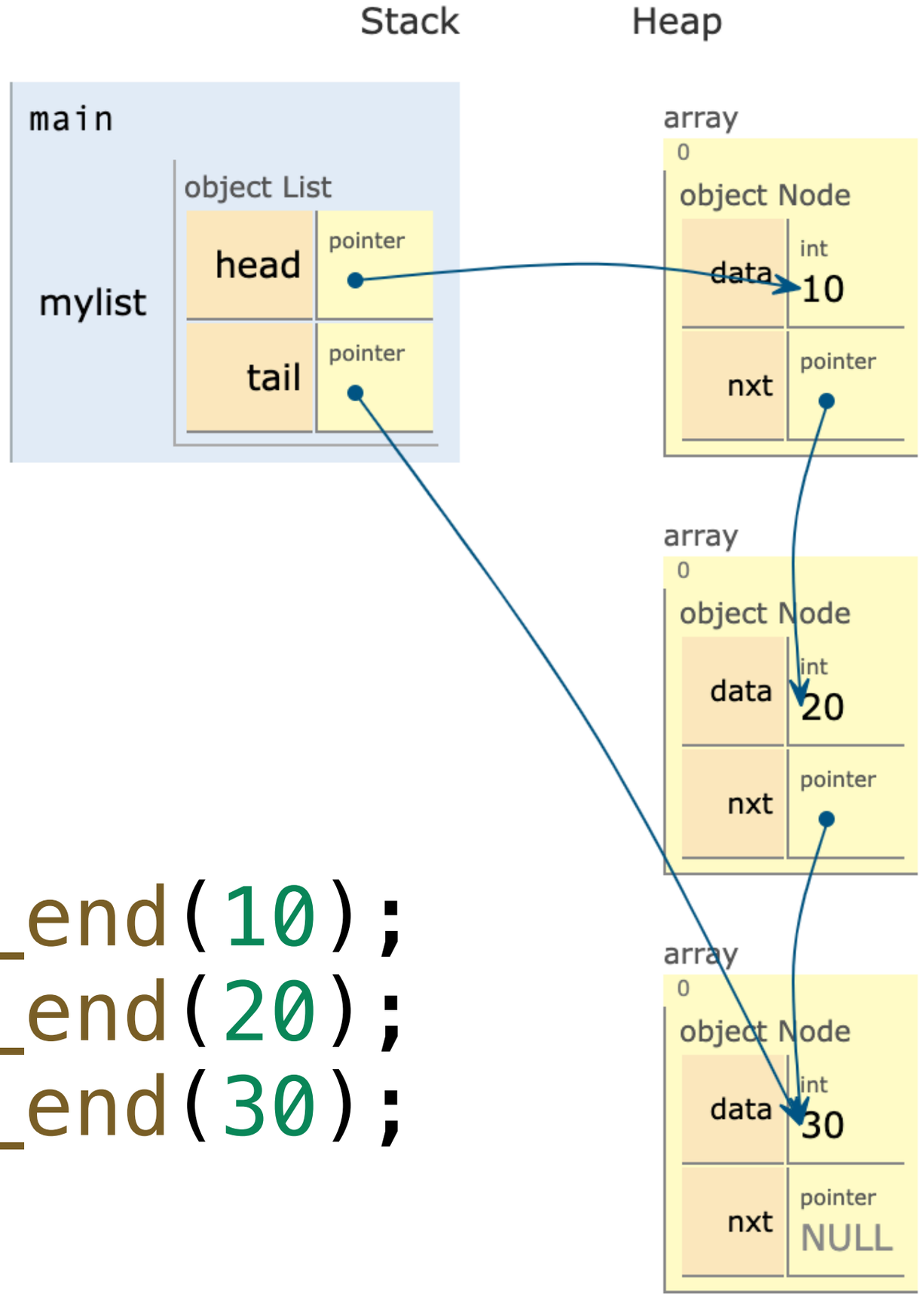
Singly Linked List

Head

(optional) Tail



NULL pointer




```
int main() {
    List mylist;
    mylist.insert_end(10);
    mylist.insert_end(20);
    mylist.insert_end(30);
}
```

credit: pythontutor.com

Operations on Linked Lists

- Linked lists are just **collections** of sequential data
  - can **insert** 1 or more elements
    - front, end, by index, by value (sorted lists)
  - can **delete** 1 or more elements
    - front, end, by index, by value
  - can **search** for a specific element
  - can **get** an element at a given index
  - can **traverse** the list
    - visit all nodes and perform an operation (e.g. print or destroy)
  - …

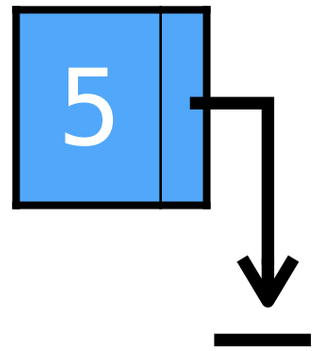
Implementing a Singly Linked List

Linked lists in C++ (prereqs)

- C++ Classes
- Pointers
  - `NULL` pointers
- Dynamic Memory Allocation
  - `new`
  - `delete`
- Pointers and Classes
  - dot notation (`.`)
  - arrow notation (`->`)

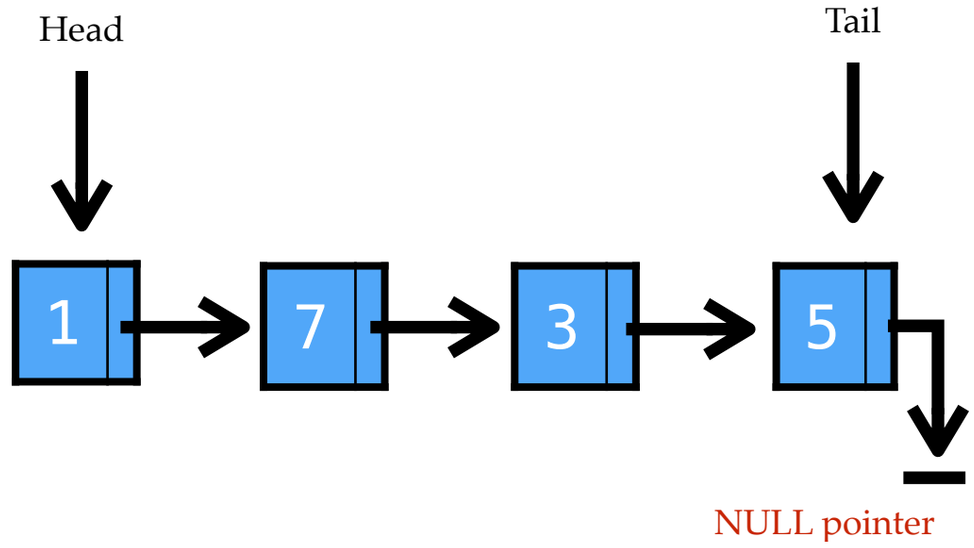
```
class Node {
    private:
        int data;
        Node *next;
    public:
        Node(int d);
        ~Node();

        friend class List;
};
```



```
class List {
    private:
        Node *head;
        Node *tail;
        // private data/methods
        // ...

    public:
        List();
        ~List();
        // public methods
        // ...
};
```



Append (insert at end)

Prepend (insert at front)

Insert by index

Delete at front

Delete at end

Delete by value

Delete by index

Get

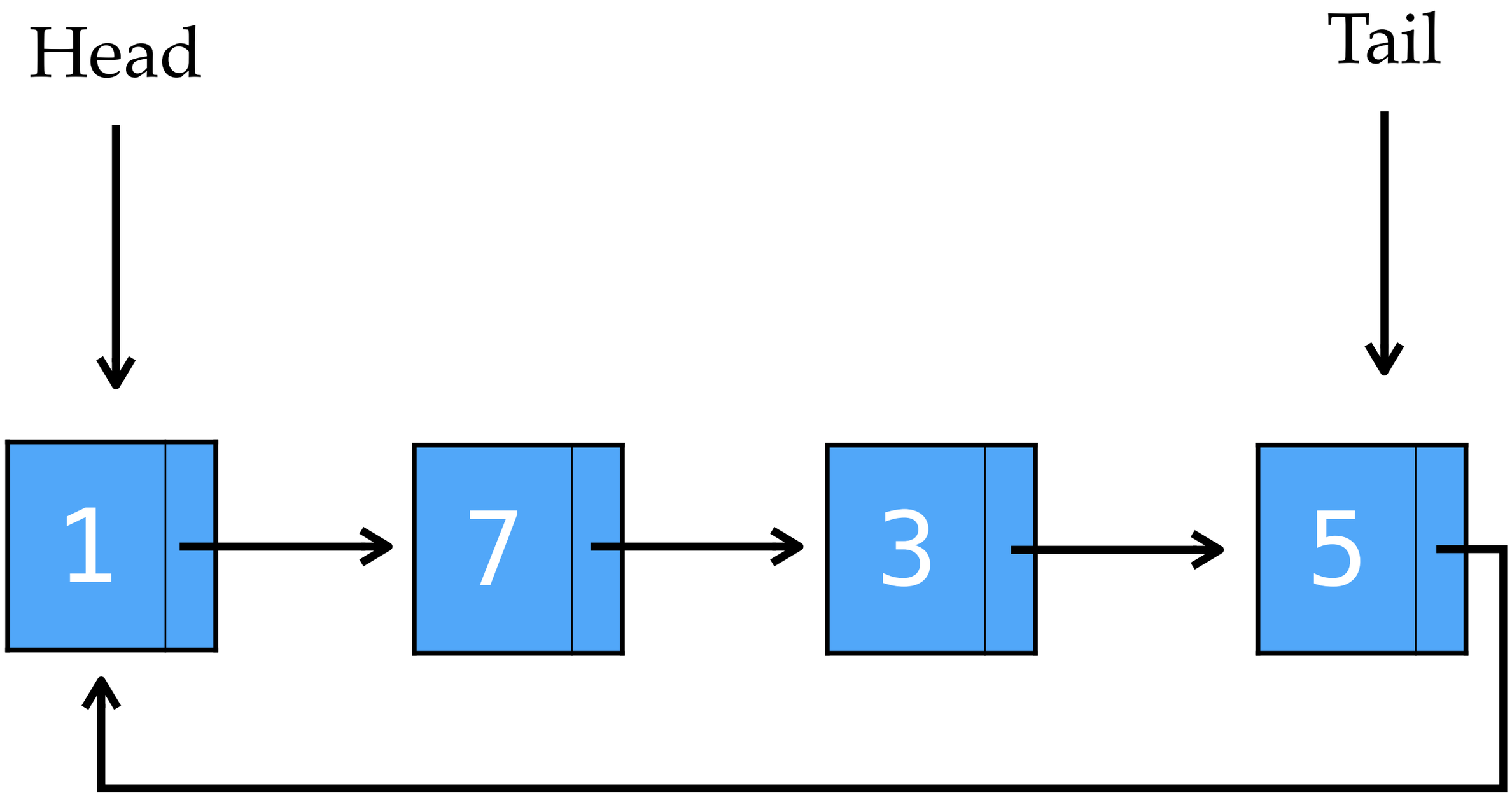
Search

Destroy (freeing a linked list)

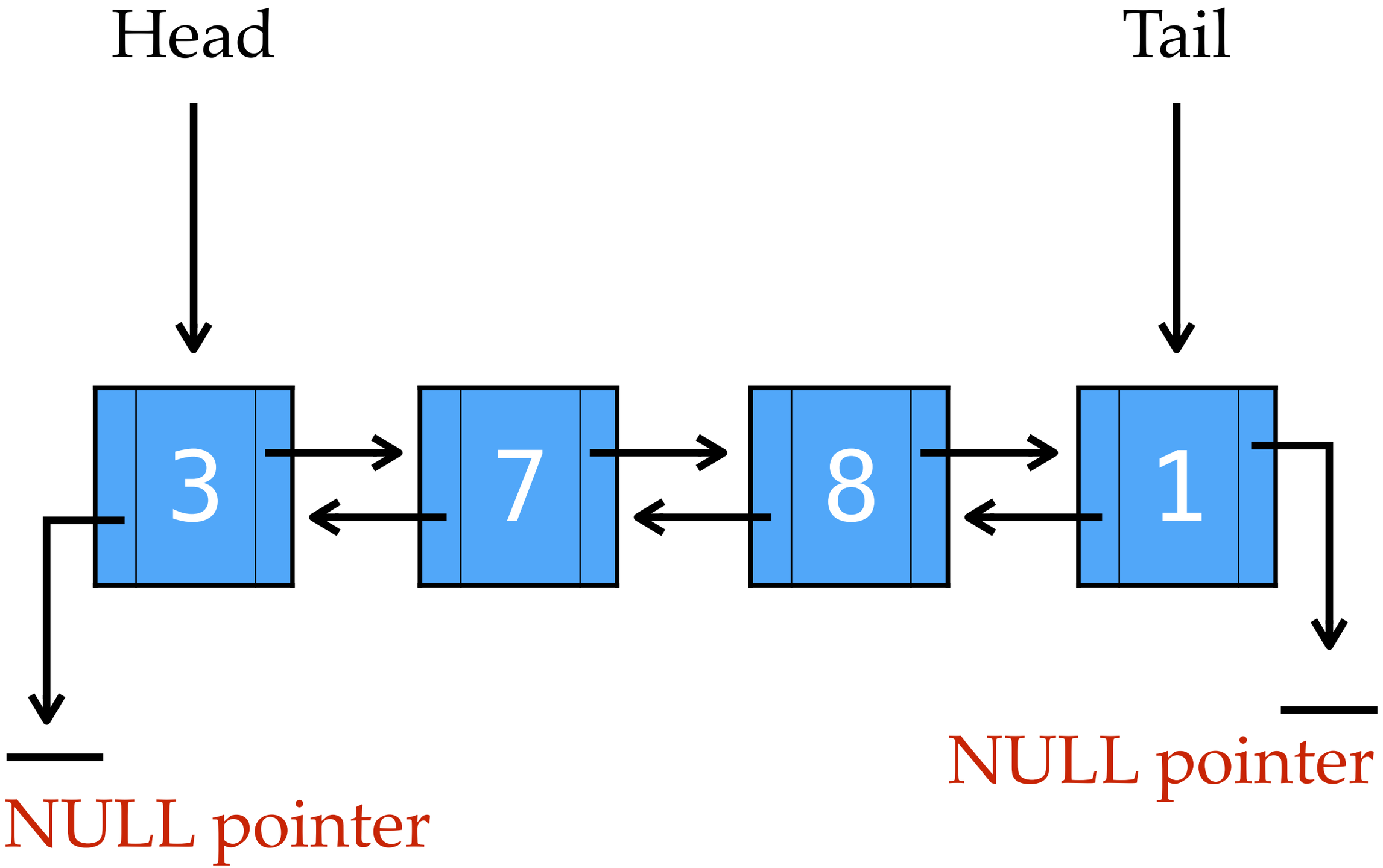
Traversing a linked list

- Using a **loop**
  - e.g. given a pointer to a starting node, prints all nodes in order

Circular Singly Linked List



Doubly Linked List



Circular Doubly Linked List