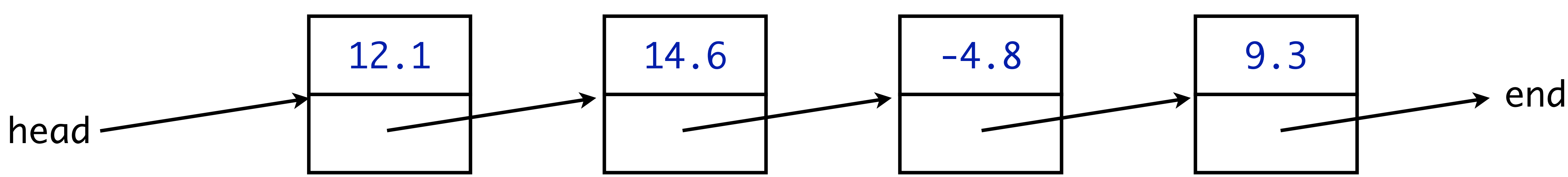
Linked Lists

Linked Lists

A linked list is a sequence of items

- each item is stored in a structure called a node
- each node is connected to another by a link (pointer)
- nodes can have links forward, backward, or both
- the last node must indicate that it is at the end (no forward link)

Simple linked list:



Nodes

Nodes are typically represented by a class:

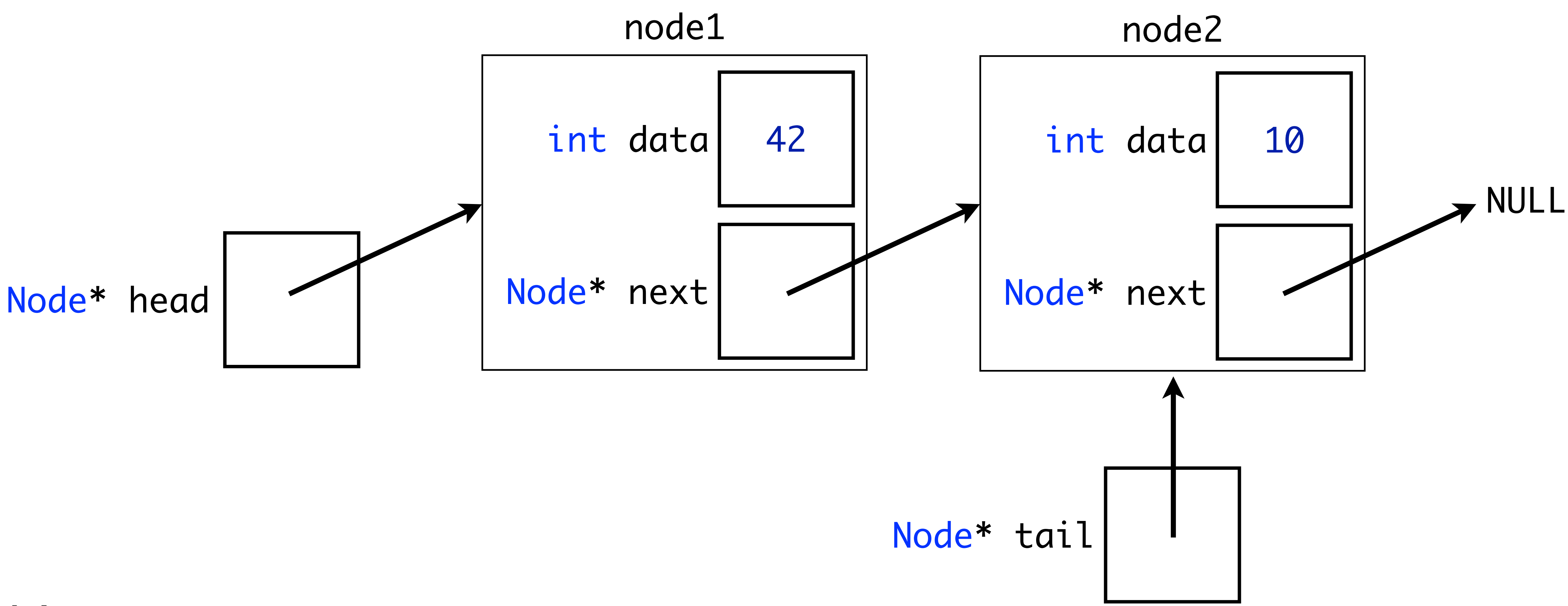
```
class Node {
    public:
        typedef ______ value_type;
    private:
        value_type data; // some data
        Node* next;      // a link to the next node
};
```

Nodes typically have at least two fields:

- a single data value
- a link (pointer) to the next node in the list

Nodes

Simple linked list of integers:

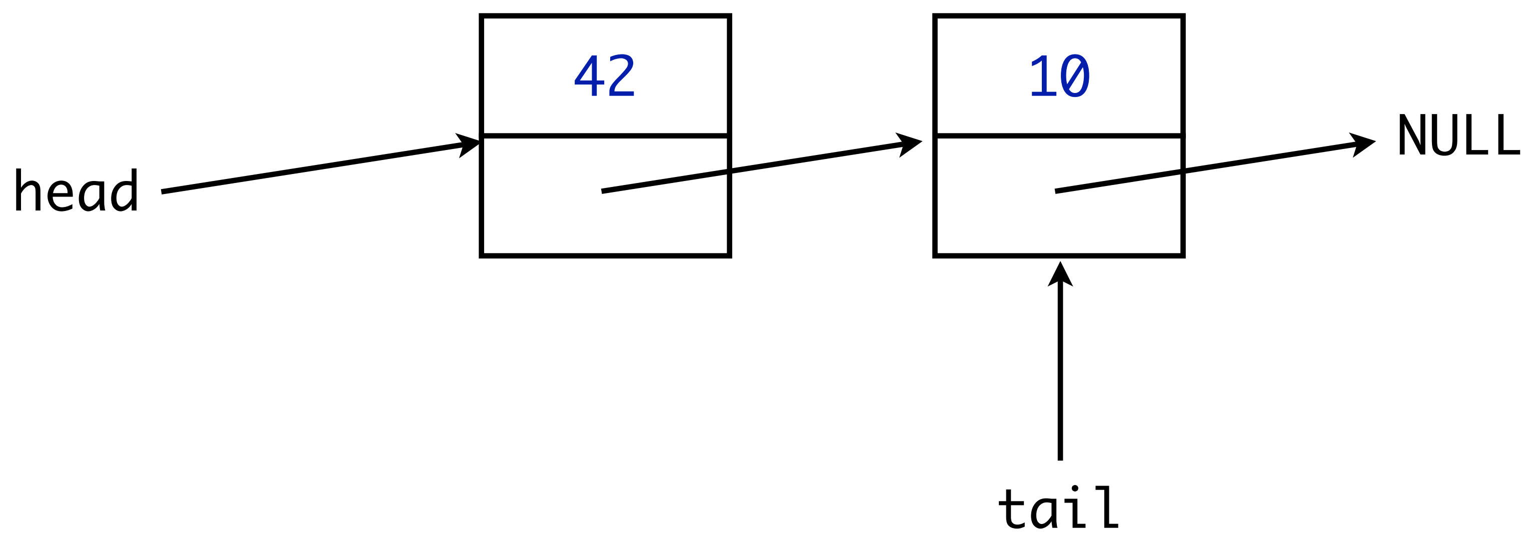


Notice:

- there is a pointer to a Node called head, which keeps track of the first node in the list
- the last node points to NULL, signifying the end of the list
- you can keep track of the end of the list, as well (the tail pointer)

Nodes

Simple linked list of integers:



A more compact representation (I'm lazy)

- there is a pointer to a `Node` called `head`, which keeps track of the first node in the list
- the last node points to `NULL`, signifying the end of the list
- you can keep track of the end of the list, as well (the `tail` pointer)

`NULL`

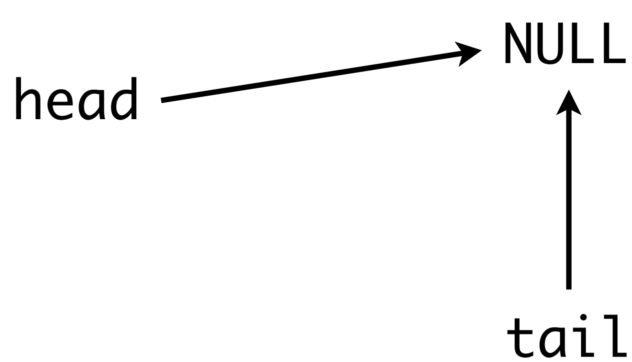
A constant defined in `csdtlib`

Used for pointers that don't point to anything

Nodes

What's it mean when when the `head` and `tail` pointers are `NULL`?

- the linked list is empty (no nodes)



`// an empty linked list`

Structs

A node can be conveniently represented as a struct:

```
struct Node {
    typedef ______ value_type;
    value_type data; // some data
    Node* next;      // a link to the next node
};
```

A struct is identical to a class in most ways...

- properties in a struct default to public, whereas those in a class default to private
- inheritance in structs defaults to public; classes default to private inheritance

Structs

Comparing struct and class:

```
struct Node {
    // structs are public by default
};

class Node {
    // classes are private by default
};
```

A struct is identical to a class in most ways...

- properties in a struct default to public, whereas those in a class default to private
- inheritance in structs defaults to public; classes default to private inheritance

Structs

You can still use visibility modifiers:

```
struct Node {
    // public by default

    private:
        // now private...
    protected:
        // now protected...
    public:
        // back to public
};
```

Structs

You can still make methods and constructors:

```
struct Node {
    // a public constructor
    Node();

    // a public method
    void do_something() const;

    // a public data member
    int data;
};
```

A struct is basically a class

But it defaults to public

Simple, right?

Nodes

Let's say we had a Node struct declared like this:

```
struct Node {
    int data;
    Node* next;
};
```

Draw a diagram for following code:

```
Node* n = new Node;
n->data = 42;
n->next = new Node;
n->next->data = 10;
n->next->next = NULL;
```

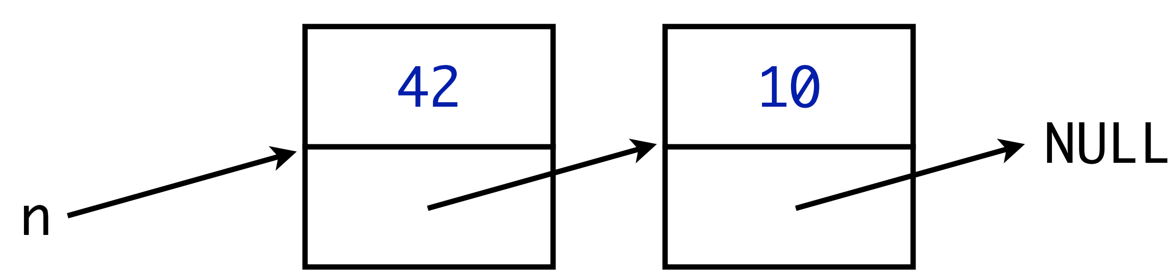
Nodes

Let's say we had a Node struct declared like this:

```
struct Node {
    int data;
    Node* next;
};
```

Draw a diagram for following code:

```
Node* n = new Node;
n->data = 42;
n->next = new Node;
n->next->data = 10;
n->next->next = NULL;
```



Nodes

More on the Node class / struct:

```
struct Node {
    typedef ______ value_type;

    // constructor with default arguments
    Node(const value_type& d = value_type(),
         Node* n = NULL): data_field(d), next(n) { }

private:
    value_type data_field; // some data
    Node* next;            // a link to the next node
};
```

Nodes

Take a closer look at the constructor:

```
// constructor with default arguments
Node(const value_type& d = value_type(),
     Node* n = NULL): data_field(d), next(n) { }
```

It provides defaults for both of its arguments

- the first uses the default constructor for `value_type` (whatever it happens to be)
- the link argument uses `NULL` as the default

When `value_type` is a built-in data type (`int`, `bool`, `double`, or `char`):

- the default value is zero (false for `bool`s)

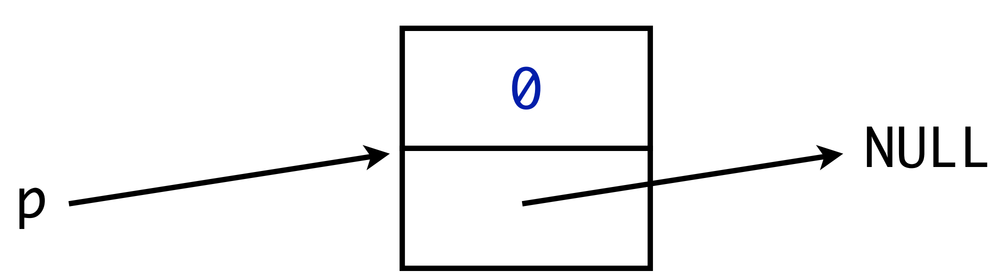
Nodes

Take a closer look at the constructor:

```
// constructor with default arguments
Node(const value_type& d = value_type(),
     Node* n = NULL): data_field(d), next(n) { }
```

We can use it to create Nodes in three different ways:

```
// default values for both data and next
p = new Node;
```



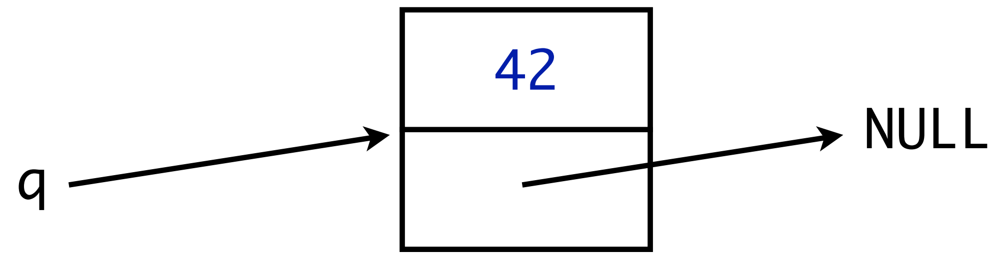
Nodes

Take a closer look at the constructor:

```
// constructor with default arguments
Node(const value_type& d = value_type(),
     Node* n = NULL): data(d), next(n) { }
```

We can use it to create Nodes in three different ways:

```
// specify the data explicitly; use NULL for link
q = new Node(42);
```



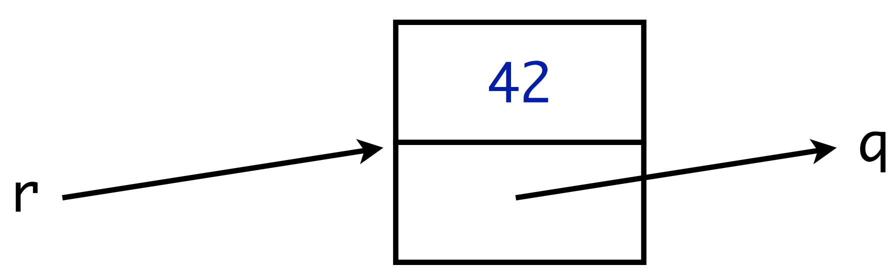
Nodes

Take a closer look at the constructor:

```
// constructor with default arguments
Node(const value_type& d = value_type(),
     Node* n = NULL): data(d), next(n) { }
```

We can use it to create Nodes in three different ways:

```
// specify both the data (42) and the link (to node q)
r = new Node(42, q);
```



Nodes

Member functions for the Node class:

```
// setter for the data field
void set_data(const value_type& new_data) {
    data_field = new_data;
}

// setter for the link field
void set_link(Node* new_link) {
    next = new_link;
}
```

Nodes

Member functions for the Node class:

```
// getter for the current data
value_type data() const {
    return data_field;
}
```

Nodes

Member functions for the Node class:

```
// getter to retrieve the Node's link
Node* link() {
    return next;
}

// ANOTHER getter to retrieve the current link
const Node* link() const {
    return next;
}
```

const and non-const methods

Why do we need two functions for the exact same task?

- one version of the method will be used by const objects
- the other will be used for non-const objects

Example:

```
// a pointer to a const node
const Node* c;
c->link(); // uses the const version of link()

// a pointer to a non-const node
Node* n;
n->link(); // uses the non-const version of link()
```

Pointers and const

const can have multiple meanings with pointers:

```
// p is a pointer to a constant Node
//  - the node CANNOT be modified via the pointer
//  - p CAN be modified to point at something else
const Node* p;
```

Examples:

```
p->data = 42; // invalid; cannot modify node via p
p = new Node; // valid; can change where p points
```

Pointers and const

const can have multiple meanings with pointers:

```
// p is a constant pointer to a Node
//   - the node CAN be modified via the pointer
//   - p CANNOT be modified to point at something else
Node* const p;
```

Examples:

```
p->data = 42; // valid; can modify node via p
p = new Node; // invalid; cannot change where p points
```

Pointers and const

const can have multiple meanings with pointers:

```
// p is a constant pointer to a constant Node
//  - the node CANNOT be modified via the pointer
//  - p CANNOT be modified to point at something else
const Node* const p;
```

Examples:

```
p->data = 42; // invalid; cannot modify node via p
p = new Node; // invalid; cannot change where p points
```

Pointers and const

Given this declaration:

```
Node node;
const Node* p1 = &node; // p1 points at node
```

You might think that `node` can never be changed...

```
p1->data = 42; // invalid; cannot modify node via p
```

However, this is not necessarily true:

```
Node* p2 = &node; // p2 also points at node...
p2->data = 42;  // and CAN change it!
```

const and non-const methods

C++ must enforce these const rules...

- so, pointers to const objects can only invoke const methods on those objects

That's why the following is true:

```
// a pointer to a const node
const Node* p;
p->link(); // uses the const version of link()

// a pointer to a non-const node
Node* n;
n->link(); // uses the non-const version of link()
```

const and non-const methods

So, we need two version of some methods (like link)

```
// non-const version
Node* link() { return next; }

// const version
const Node* link() const { return next; }
```

Why?

- the non-const version enables `link` to be used in methods that modify the list
- the const version enables `link` to be used on pointers to const objects

Linked List Functions

Linked List Functions

Determine the length of a linked list:

```
// returns the number of nodes in a linked list
size_t list_length(const Node* head_ptr);

// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   the value returned is the number of nodes in the
//   linked list
```

Linked List Functions

Insert an item at the front of a list:

```
// inserts @entry at the beginning of @head_ptr's list
void list_head_insert(Node*& head_ptr,
                      const Node::value_type& entry);


// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   a new node containing the given entry has been
//   added at the head of the linked list; head_ptr now
//   points to the head of the new, longer linked list
```

Linked List Functions

Insert an item after a node in a list:

```
// inserts @entry after @previous_ptr in a list
void list_insert(Node* previous_ptr,
                 const Node::value_type& entry);

// precondition:
//   previous_ptr points to a node in a linked list
// postcondition:
//   a new node containing the given entry has been
//   added after the node pointed at by previous_ptr
```

Linked List Functions

Search for an item in a list (non-const version):

```
// returns a pointer to @target if it's in a linked list
Node* list_search(Node* head_ptr,
                  const Node::value_type& target);

// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   the pointer returned points to the first node
//   containing the specified target in its data member.
//   If there is no such node, NULL is returned
```

Linked List Functions

Search for an item in a list (const version):

```
// returns a pointer to @target if it’s in a linked list
const Node* list_search(const Node* head_ptr,
                        const Node::value_type& target);

// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   the pointer returned points to the first node
//   containing the specified target in its data member.
//   If there is no such node, NULL is returned
```

Linked List Functions

Search for an item at a specific location in a list (non-const version):

```
// returns the item at @position in a linked list
Node* list_locate(Node* head_ptr,
                  size_t position);


// precondition:
//   head_ptr is the head pointer of a linked list, and
//   position is greater than 0
// postcondition:
//   the pointer returned points to the node at the
//   specified position in the list (starting at 1). If
//   there is no such position, then NULL is returned
```

Linked List Functions

Search for an item at a specific location in a list (const version):

```
// returns the item at @position in a linked list
const Node* list_locate(const Node* head_ptr,
                        size_t position);

// precondition:
//   head_ptr is the head pointer of a linked list, and
//   position is greater than 0
// postcondition:
//   the pointer returned points to the node at the
//   specified position in the list (starting at 1). If
//   there is no such position, then NULL is returned
```

Linked List Functions

Removes the node at the head of a list:

```
// removes the node at the head of a linked list
void list_head_remove(Node*& head_ptr);

// precondition:
//   head_ptr is the head pointer of a linked list, with
//   at least one node
// postcondition:
//   the head node has been removed and returned to the
//   heap; head_ptr is now the head pointer of the new,
//   shorter linked list
```

Linked List Functions

Removes the node after the specified node:

```
// removes the node following @previous_ptr in a list
void list_remove(Node* previous_ptr);

// precondition:
//   previous_ptr points to a node in a linked list and
//   is not the tail node of the list
// postcondition:
//   the node after previous_ptr has been removed from
//   the linked list
```

Linked List Functions

Clears the linked list:

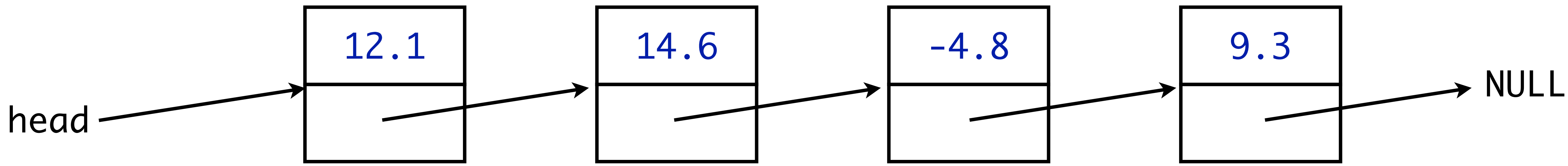
```
// clears the linked list identified by @head_ptr
void list_clear(Node*& head_ptr);


// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   all nodes of the list have been returned to the
//   heap, and the head_ptr is now NULL
```

That was fun.

Traversing a Linked List

How would you iterate over this linked list?



Start at the beginning and go till the end!

```
for (Node* n = head; n != NULL; n = n->link()) {
    cout << n->data() << endl;
}
```