CS 112

Introduction to Data Structures

WEEK 06

Linked Lists

Eric Araújo

Calvin University · Fall 2026

This Week

1. What dynamic arrays are bad at
2. Nodes and links
3. The `Node` and `List` classes
4. Traversing
5. `prepend` and `append`
6. The destructor chain reaction **KEY**
7. Linked list vs. dynamic array

Where Dynamic Arrays Hurt

- `append()` when full: allocate, copy everything, delete
- `insert()` in the middle: shift every element after it
- `remove()`: shift everything back down
- Growing needs one **contiguous** block big enough for all of it

> All four problems come from the same source: the elements must sit *next to each other* in memory.

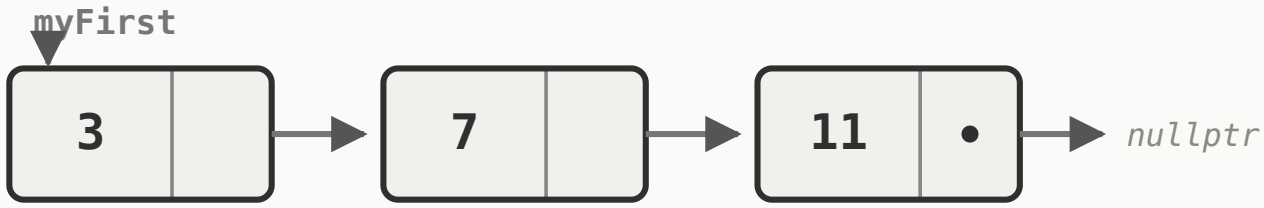
What If We Broke It Up?

Store the data in chunks instead of one block. Inserting then only shifts within a chunk.

- Chunks of 5? 3? 2?
- What happens when a chunk fills up?
- …what if each chunk held exactly **one** item?

> Then nothing ever shifts. But if the items aren't neighbours in memory, how do we find the next one? **Each one has to point to it.**

A Linked List



Each **Node** holds one item and the address of the next Node. The last one points to `nullptr`, which is how you know you've reached the end.

Two Classes

Node: one link

```
class Node {
 public:
  Node();
  Node(Item it, Node *nxt);
  ~Node();
  Item  myItem;
  Node *myNext;
};
```

List: the container

```
class List {
 public:
  List();
  ~List();
  void prepend(Item it);
  void append(Item it);
 private:
  unsigned mySize;
  Node *myFirst;
  Node *myLast;
};
```

> Students of your class only ever see `List`. `Node` is an implementation detail, which is exactly what `private` is for.

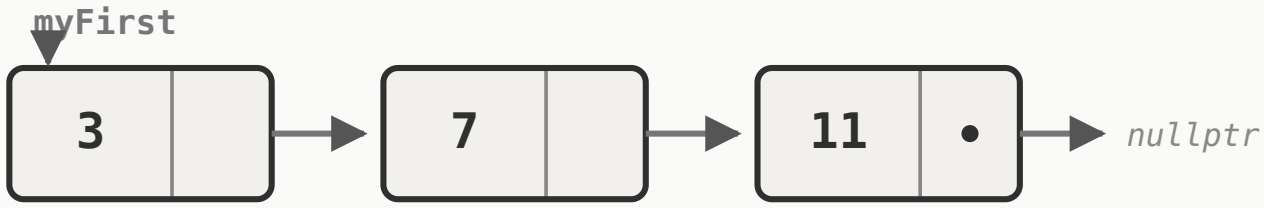
Traversing

```
Node *current = myFirst;

while (current != nullptr) {
    cout << current->myItem << endl;
    current = current->myNext;     // hop to the next
}
```

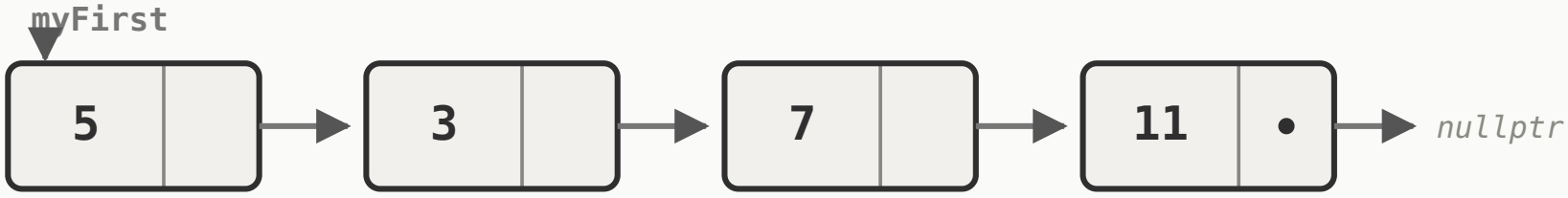
This loop is the linked-list equivalent of `for (i = 0; i < n; ++i)`. You will write it dozens of times, and every one of its bugs comes from forgetting the `nullptr` test or forgetting to advance.

prepend(5)



> Adding at the front. What has to change?

prepend(5)




```
void List::prepend(Item it) {
    myFirst = new Node(it, myFirst);   // point the new one at the old first
    if (mySize == 0) { myLast = myFirst; }
    ++mySize;
}
```

> Three steps, no shifting, no matter how long the list is, **constant time**.

TALK TO YOUR NEIGHBOR · TTYN

Why does `prepend()` need that `if (mySize == 0)`?

A. To avoid a memory leak

B. Because on an empty list the new node is also the *last* node

C. To keep `mySize` correct

D. It isn't needed, `myLast` is never used

append(), Why Keep `myLast`?

```
void List::append(Item it) {
    Node *n = new Node(it, nullptr);
    if (mySize == 0) {
        myFirst = myLast = n;
    } else {
        myLast->myNext = n;   // old last points at n
        myLast = n;
    }
    ++mySize;
}
```

Without `myLast` you'd walk the whole list to find the end, linear. Storing one extra pointer makes `append()` constant. **A little memory buys a lot of speed.**

The Destructor Chain Reaction

```
Node::~Node() {
    delete myNext;      // destroy the rest of the list
}

List::~List() {
    delete myFirst;     // starts the chain
}
```

Deleting the first Node runs *its* destructor, which deletes the second, which deletes the third…

> `delete nullptr` is harmless and defined; that's what stops the chain at the end. Elegant, but on a very long list it also means a very deep stack of destructor calls.

TALK TO YOUR NEIGHBOR · TTYN

A `List` holding 4 items goes out of scope. How many `Node` destructors run?

- **A.** 1, only the first node is deleted
- **B.** 4, each node deletes the next
- **C.** 0, the compiler handles it
- **D.** 5, including the nullptr at the end

Demo

Walking a list, one hop at a time.

(the list is fine. the loop is forever.)

Linked List vs. Dynamic Array

Operation	Dynamic array	Linked list
index `v[i]`	**constant**	linear
append	amortized constant	**constant**
prepend	linear	**constant**
insert in middle	linear	linear to find, constant to link
remove	linear	linear to find, constant to unlink
traverse all	linear	linear
memory per item	just the item	item + a pointer

> Neither wins. The array is built for *reaching*, the list for *rearranging*.

TALK TO YOUR NEIGHBOR · TTYN

You need a collection you'll mostly add to at the front and read in order. Which?

A. Dynamic array, indexing is faster

B. Linked list, prepending is constant and you never index

C. Either, they're the same in practice

D. Dynamic array, because it uses less memory per item

Week 06 Recap

- A linked list trades **contiguity** for **flexibility**
- Each `Node` holds an item and the address of the next
- Traversal is `while (current != nullptr)`: advance or loop forever
- `prepend` and `append` are constant; indexing is linear
- The empty list and the one-item list are where the bugs hide
- Node destructors chain, and `delete nullptr` ends the chain safely

Next: a07 →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**