Pre-Class Work 5: Linked Nodes

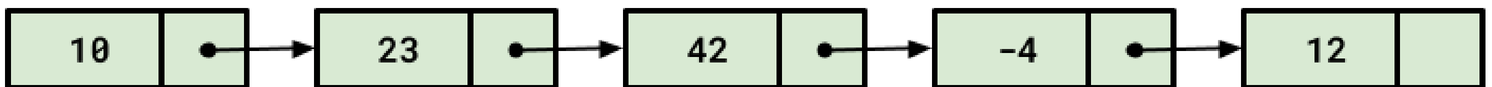
Non-Contiguous Memory

So far, we have primarily seen how to use arrays to store a list of values using **contiguous memory**. This means that the space taken up by the array in your computer is all in one large block. The elements end up side-by-side in memory.

This structure is nice to randomly access elements, since we know exactly where to find it in the block of memory for the array. Commonly we picture arrays like so:

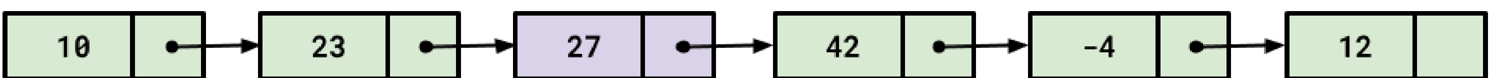
Instead of this, an alternative is **non-contiguous memory**. Here, the elements of our list are scattered throughout memory, we cannot quickly find a random element like we could with our blocks of memory.

Rather, we need some way for each element to "link" to the next element to continue the list. You could imagine our elements now look like this:



We'll learn more about some of the advantages and disadvantages of these linked structures in future weeks, but for now think about how much easier it is to insert something in the middle of this list than an array.

Rather than shifting all the other elements to the left or right, we can just change two links to add an element in the middle!



Linked nodes are an alternative to arrays for representing an ordered sequence of elements. Whereas arrays store elements side-by-side in memory, linked nodes store each element in their own "node" object and "link" them together using references.

🔑 Main Points

- *Arrays* use **contiguous memory**, meaning they store elements side-by-side in memory.
- **Linked nodes** are an alternative to arrays that store each element in their own "node" object and "link" them together using references.
- This is referred to as **non-contiguous memory**. We cannot quickly find a random element like we can in arrays because elements of our list are scattered throughout memory.
- It's easier to insert something in the middle of a list than in an array! We will learn more advantages and disadvantages of lists later.

ListNode Class

Remember learning about `Queue`s in CSE 122 and how we used the `LinkedList` implementation? This week we'll dive into how exactly a `LinkedList` works by implementing our own version, `LinkedIntList`. We'll dive deeper into `LinkedIntList` the rest of this week, but for now we'll work with individual nodes like we described in the previous slide.

To represent these "linked" elements, we will use linked nodes. Each linked node stores its element data and a "link" to the next node using references.

For example, we will use the following `ListNode` class to represent linked nodes. The `ListNode` class contains 2 fields:

- The value of this node's element as the `data` field.
- A reference to a `next` node.

```
// Class that represents a single node containing an
// integer value.
public class ListNode {
    public final int data;
    public ListNode next;

    // Constructs a ListNode with the given data
    public ListNode(int data) {
        // Sets the next field to null, meaning there
        // is no next linked node.
        this(data, null);
    }

    // Constructs a ListNode with the given data
    // and given next node.
    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```

It might feel a bit weird for the `ListNode` class to contain a field referencing itself, but this is allowed in Java. We use the `next` field to access the node in the list after the current one.

Remember that this is only a **reference** and does not store the actual object itself. We'll see some examples of how to use these in the next slide.

The `data` field is `final` to prevent changes to `data` after the `ListNode` is created. In this class, we'll focus on changing the value of the `next` field in order to solidify our understanding of reference semantics.

Note that the fields of the `ListNode` class are declared `public`. In most situations, this would allow clients to modify the `ListNode` values and break the class invariants needed to ensure the program works correctly! But we'll later learn ways to hide the `ListNode` implementation from clients.

`null` Review

You may have seen `null` as a programmer already. `null` is defined as the "absence of a value", it is treated as a special term by programmers to convey that there is no value. We use it frequently with linked nodes to represent the end of our list.

Main Points

- We use **linked nodes** to build linked data structures like `LinkedLists`. Each linked node has two fields: the `data` field, which stores the value of the node, and the `next` field, which is a reference to the next node in the sequence.
- Fields that are objects, like `next`, store a reference to an object in memory rather than the object itself.
- : The **`ListNode`** class represents a single node in a linked list and has two constructors:
  - One that constructs a node with just given data (the next node is null).
  - One that constructs a node with given data *and* a node that comes next.
- We declared the `data` field as `final` so we cannot change the data once we make a `ListNode` object.
- We use `null` frequently with linked nodes to represent the end of our list.

ListNode Review

ListNode Review

These questions are designed to assess your current understanding of linked nodes. It is highly recommended that you draw out what the result of the code below would look like. For all the questions below assume the following piece of code has been run:

```
public class Example {
    public static void main(String[] args) {
        ListNode a = new ListNode(6);
        ListNode b = new ListNode(15);
        ListNode c = new ListNode(-3, b);
        ListNode d = new ListNode(14, c);
        a.next = d;

        ListNode temp = a;
        while (temp != null) {
            System.out.println(temp.data);
            temp = temp.next;
        }
    }


    // The ListNode class we showed in the previous slide.
    // WARNING: IGNORE STATIC HERE, your Node classes should be in their own file.
    //          This is just a hack to run on Ed.
    // Class that represents a single node containing an
    // integer value.
    public class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Question 1

What would `System.out.println(a.data)` output?

- ◯ 14
- ◯ 6
- ◯ -3
- ◯ 15

Question 2

What would `System.out.println(a.next.next.data)` output?

- ◯ 14
- ◯ 6
- ◯ -3
- ◯ 15

Question 3

Which of the following produce the value 15?

- ◯ `a.data`
- ◯ `a.next.data`
- ◯ `a.next.next.data`
- ◯ `a.next.next.next.data`

ListNode Programming Review

Modify the given program so that the list currently in the form `a -> [1, 2, 4]`, instead looks like `a -> [1, 2, 3, 4]`.

You should only create **one** new `ListNode` to do this. You should not modify the `data` field of any existing node and should instead modify the `next` fields.

Implementation Hints:

- How many `.next`s from `a` is the 4 node?
- What `.next`s do you need to change?