Lec 8: Doubly Linked Lists

Sam McCauley

February 25, 2026

Admin

- In-person lab today
  - Starter code is: `ArrayList.java`, `NodeInt.java`, `LinkedListInt.java` plus files from today

- Any questions?

Linked List of Ints

Int Linked List Nodes (Review)

- In our example, students held the information of the list
- We'll store the information in `Nodes`
    - We'll call them `NodeInt` to clarify that these nodes only help us to store lists of `ints`
- What should the `NodeInt` class look like? Let's code it up together.

Int Linked List Class

- The list object only needs to store a reference to the first node!

Int Linked List Class

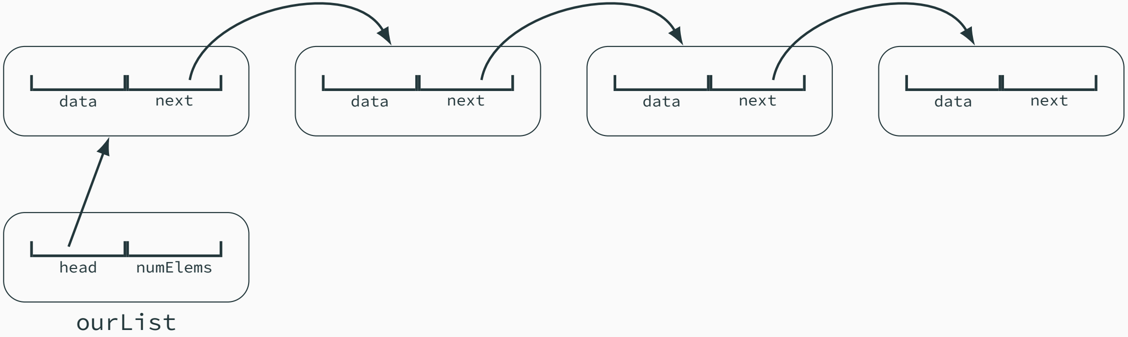
- The list object only needs to store a reference to the first node!

- We'll also store the number of elements in the list again, like we did for IntArrayList (will be handy)

Int Linked List Class

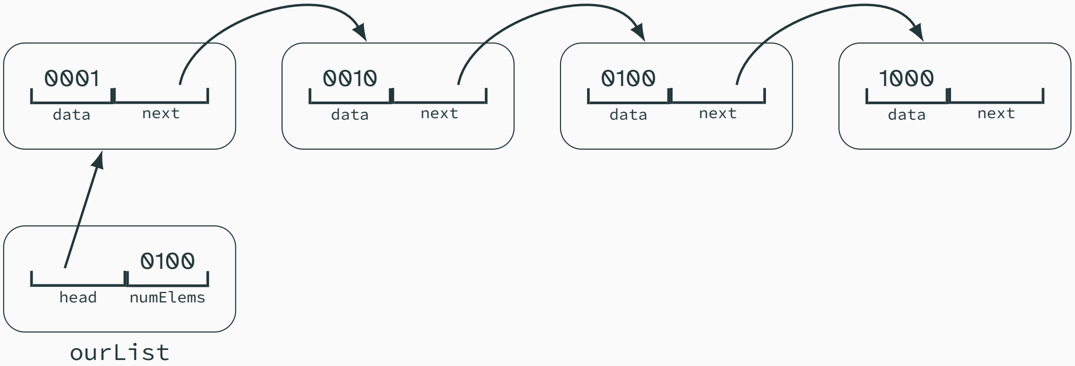
- The list object only needs to store a reference to the first node!
- We'll also store the number of elements in the list again, like we did for IntArrayList (will be handy)
- Maintain the following invariant: the `next` pointer of a node is `null` only when the node is the last in the list

Overview of Plan

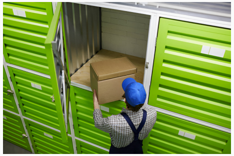


Overview of Plan: With Data

The following list stores: [1, 2, 3, 4]. `int`s are truncated to 4 bits for readability.

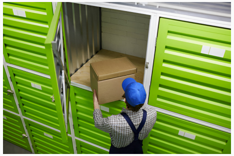


get and set



- `int get(int index)` returns element `index` stored in the linked list. We'll start by accessing the Node stored in `head`; its value is correct for index `0`. Then, we'll go to the `next` element of that Node, then its next, etc., `index` times. Don't forget to do bounds checking!

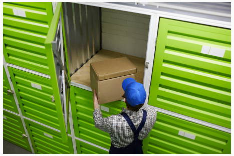
get and set



- `int get(int index)` returns element `index` stored in the linked list. We'll start by accessing the Node stored in `head`; its value is correct for index `0`. Then, we'll go to the `next` element of that Node, then its next, etc., `index` times. Don't forget to do bounds checking!

- `int set(int index, int newElement)` sets the element stored at `index` to be `newElement`, and returns the element *previously* stored in slot `index`. We'll accomplish this similarly: scan through the list until the ith node, but now we set its value rather than getting it.

get and set



- `int get(int index)` returns element `index` stored in the linked list. We'll start by accessing the Node stored in `head`; its value is correct for index `0`. Then, we'll go to the `next` element of that Node, then its next, etc., `index` times. Don't forget to do bounds checking!

- `int set(int index, int newElement)` sets the element stored at `index` to be `newElement`, and returns the element *previously* stored in slot `index`. We'll accomplish this similarly: scan through the list until the ith node, but now we set its value rather than getting it.

- Both methods need to scan through `index` nodes. Let's set up a method to do that!

add

- `void add(int newElement)` adds `newElement` to the end of the list. (Fun video)

add

- `void add(int newElement)` adds `newElement` to the end of the list. (Fun video)

- To make a new element, we’ll need a new place to store it: a new `NodeInt`.

add

- `void add(int newElement)` adds `newElement` to the end of the list. (Fun video)
- To make a new element, we’ll need a new place to store it: a new `NodeInt`.
- Then, we need to add it to the end of the list. To do that, we first need to find the end of the current list.

add

- `void add(int newElement)` adds `newElement` to the end of the list. (Fun video)
- To make a new element, we'll need a new place to store it: a new `NodeInt`.
- Then, we need to add it to the end of the list. To do that, we first need to find the end of the current list.
- Finally, we need to "join" our new node to the end of the list: the `next` value of the node at the end of the list should be the new node we made.

add

- `void add(int newElement)` adds `newElement` to the end of the list. (Fun video)
- To make a new element, we'll need a new place to store it: a new `NodeInt`.
- Then, we need to add it to the end of the list. To do that, we first need to find the end of the current list.
- Finally, we need to "join" our new node to the end of the list: the `next` value of the node at the end of the list should be the new node we made.
- Let's do this in the code

Adding at an index

- `void add(int index, int newElement)`: adds `newElement` to index `index` in the list, pushing all later elements in the list down by one. In a linked list, we just place the new node in the correct location. We do not need to manually “push down” elements: all later elements will automatically be one further down in the list, since there is now a new node in front of them.

Adding at an index

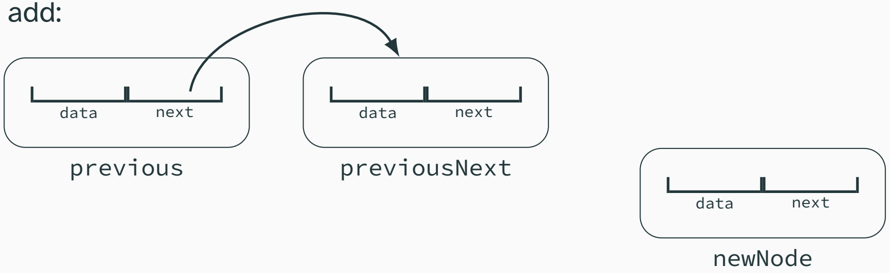
- `void add(int index, int newElement)`: adds `newElement` to index `index` in the list, pushing all later elements in the list down by one. In a linked list, we just place the new node in the correct location. We do not need to manually “push down” elements: all later elements will automatically be one further down in the list, since there is now a new node in front of them.
- This method requires some careful “surgery” on the linked list: we need to disconnect the reference to insert the new node. Let’s draw a picture of what that looks like. Let’s say we have found the $(\texttt{index} - 1)$-st node in our list; its next node is the current `index`-th node. We want our new node to go in between them.

Adding at an index

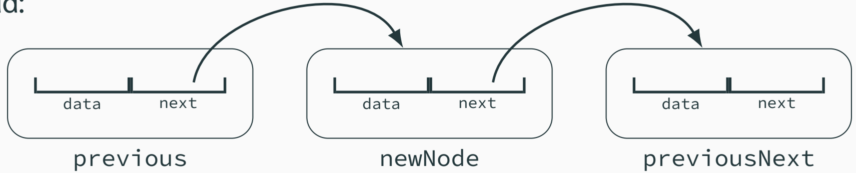
- `void add(int index, int newElement)`: adds `newElement` to index `index` in the list, pushing all later elements in the list down by one. In a linked list, we just place the new node in the correct location. We do not need to manually “push down” elements: all later elements will automatically be one further down in the list, since there is now a new node in front of them.
- This method requires some careful “surgery” on the linked list: we need to disconnect the reference to insert the new node. Let’s draw a picture of what that looks like. Let’s say we have found the $(\texttt{index} - 1)$-st node in our list; its next node is the current `index`-th node. We want our new node to go in between them.
- (Diagram on next slide)

Linked List Surgery

Before add:



After add:



More on Memory

Arrays

- Stored like objects! The variable stores a *reference* to the actual array slots

Arrays

- Stored like objects! The variable stores a *reference* to the actual array slots

- Let’s look back at the `expandCapacity` method of `ArrayListInt.java`

`this` keyword

- The `this` keyword gives a reference to the current object

`this` keyword

- The `this` keyword gives a reference to the current object
- When we use it to access instance variables, it's like "giving directions"

`this` keyword

- The `this` keyword gives a reference to the current object
- When we use it to access instance variables, it’s like “giving directions”
- Also used in `CoinStrip.java`

```java
public class Student{
    String name;
    int graduationYear;

    public Student(String name, int graduationYear) {
        //this.name is instance variable; name is parameter
        this.name = name;
        this.graduationYear = graduationYear;
    }
}
```

Multiple References to the Same Object

In pairs: what happens in this example?

```java
Student s1 = new Student("Victoria", 2005);
Student s2 = new Student("Victoria", 2005);  //s2 references a new student with the same data
System.out.println(s2.getGraduationYear()); //prints?
System.out.println(s1.getGraduationYear()); //prints?
s2.setGraduationYear(1997);
System.out.println(s1.getGraduationYear()); //prints?
System.out.println(s2.getGraduationYear()); //prints?
```

Multiple References to the Same Object

In pairs: what happens in *this* example? (Second line changed!)

```java
Student s1 = new Student("Freida",2005);
Student s2 = s1;
System.out.println(s2.getGraduationYear()); //prints?
System.out.println(s1.getGraduationYear()); //prints?
s2.setGraduationYear(1997);
System.out.println(s1.getGraduationYear()); //prints?
System.out.println(s2.getGraduationYear()); //prints?
```

Multiple References to the Same Object

```java
Student s1 = new Student("Freida",2005);
Student s2 = s1;  //s2 and s1 reference the same place
System.out.println(s2.getGraduationYear()); //prints 2005
System.out.println(s1.getGraduationYear()); //prints 2005
s2.setGraduationYear(1997);
System.out.println(s1.getGraduationYear()); //prints 1997
System.out.println(s2.getGraduationYear()); //prints 1997
```

`.equals()`

- Special Java method for objects

`.equals()`

- Special Java method for objects
- Always test equality of objects using `.equals()`

`.equals()`

- Special Java method for objects
- Always test equality of objects using `.equals()`
  - If you use `==` it tests if the *references* are the same

`.equals()`

- Special Java method for objects
- Always test equality of objects using `.equals()`
    - If you use `==` it tests if the *references* are the same
    - Need to fill in yourself when you create a class (we’ll come back to this)

`.equals()`

- Special Java method for objects
- Always test equality of objects using `.equals()`
  - If you use == it tests if the *references* are the same
  - Need to fill in yourself when you create a class (we'll come back to this)
- Always use `.equals()` for `Strings`!

```
if(string1.equals(string2)) {
    //string 1 and string 2 are equal
}
if(string1 == string2) {
    //if string 1 and string 2 are same, usually get here (
        but not always)
    //don't use this!
}
```

`.toString()`

- Second special Java method for objects

`.toString()`

- Second special Java method for objects
- Outputs a string for the object

`.toString()`

- Second special Java method for objects
- Outputs a string for the object
- Called automatically! Especially by `System.out.println()`

`.toString()`

- Second special Java method for objects
- Outputs a string for the object
- Called automatically! Especially by `System.out.println()`
- Let’s look at the main method of `LinkedListInt.java`

Doubly Linked List

- Modification to the linked list from last time

Doubly Linked List

- Modification to the linked list from last time
    - What we implmented Monday in `LinkedListInt.java`is a “Singly Linked List”

Doubly Linked List

- Modification to the linked list from last time
    - What we implmented Monday in `LinkedListInt.java`is a “Singly Linked List”
- In a singly linked list, you can only go forward to the next node; can’t go backwards

Doubly Linked List

- Modification to the linked list from last time
    - What we implmented Monday in `LinkedListInt.java`is a “Singly Linked List”
- In a singly linked list, you can only go forward to the next node; can’t go backwards
- To `add()` to the end need to traverse the entire list!

Doubly Linked List

- Modification to the linked list from last time
    - What we implmented Monday in `LinkedListInt.java`is a “Singly Linked List”
- In a singly linked list, you can only go forward to the next node; can’t go backwards
- To `add()` to the end need to traverse the entire list!
- In a Doubly Linked List, we add in the capability to go forwards or backwards

Doubly Linked List Changes

- Each node will keep track of the next node, *and* the previous node

Doubly Linked List Changes

- Each node will keep track of the next node, *and* the previous node
- The list will keep track of the first element (`head`), and the last element (`tail`)

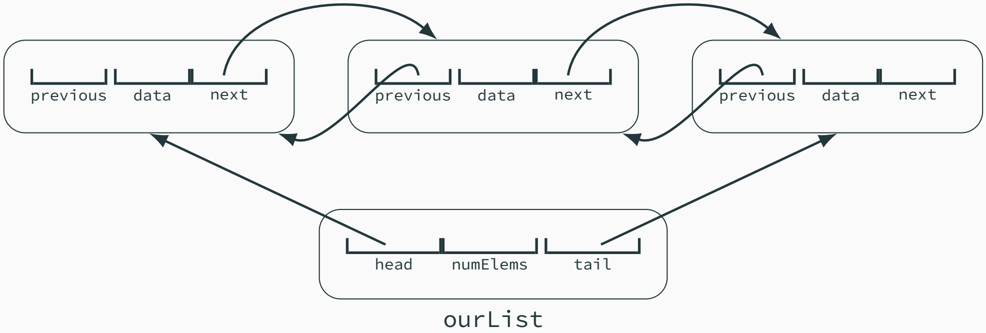
Doubly Linked List Changes

- Each node will keep track of the next node, *and* the previous node
- The list will keep track of the first element (`head`), and the last element (`tail`)
- `head`'s previous reference and `tail`'s next reference will be `null`

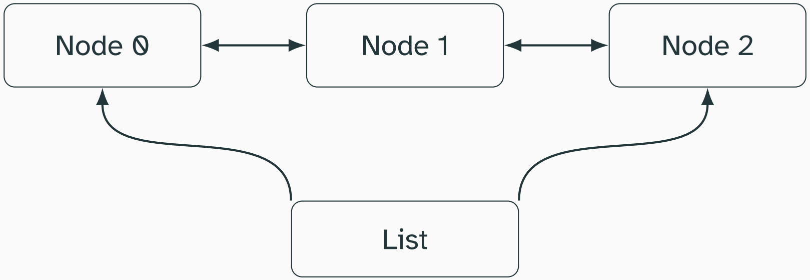
Doubly Linked List Changes

- Each node will keep track of the next node, *and* the previous node
- The list will keep track of the first element (`head`), and the last element (`tail`)
- `head`'s previous reference and `tail`'s next reference will be `null`
- If the list has one element, `head` and `tail` are the same

Doubly Linked List



Doubly Linked List: Simplified Diagram



Doubly Linked List: Some Code

- Let's look at `DLLNodeInt.java`

Doubly Linked List: Some Code

- Let’s look at `DLLNodeInt.java`
- How can we fill in `add()` (at the end of the list)?

Doubly Linked List: Some Code

- Let’s look at `DLLNodeInt.java`
- How can we fill in `add()` (at the end of the list)?
    - Let’s do this in pairs if it’s before 9:35

Doubly Linked List: Some Code

- Let’s look at `DLLNodeInt.java`
- How can we fill in `add()` (at the end of the list)?
    - Let’s do this in pairs if it’s before 9:35
- How can we fill in our private method `getNode()`?

Doubly Linked List: Some Code

- Let's look at `DLLNodeInt.java`
- How can we fill in `add()` (at the end of the list)?
    - Let's do this in pairs if it's before 9:35
- How can we fill in our private method `getNode()`?
    - Do we want to traverse the list forward or backward?

Comparing Efficiency

What We Mean By Efficiency

- We’ll talk soon about how to analyze efficiency rigorously

What We Mean By Efficiency

- We’ll talk soon about how to analyze efficiency rigorously
- For today: does the method need to traverse the entire data structure?

What We Mean By Efficiency

- We’ll talk soon about how to analyze efficiency rigorously
- For today: does the method need to traverse the entire data structure?
- If your list has millions or billions of elements, traversing the data structure takes lots of time

What We Mean By Efficiency

- We’ll talk soon about how to analyze efficiency rigorously
- For today: does the method need to traverse the entire data structure?
- If your list has millions or billions of elements, traversing the data structure takes lots of time
- Example: `indexOf()` always traverses the whole data structure. It’s a relatively slow operation

Efficiency of Methods

- Let’s compare how different data structures implement `get()`

Efficiency of Methods

- Let's compare how different data structures implement `get()`
- What about `set()`?

Efficiency of Methods

- Let’s compare how different data structures implement `get()`
- What about `set()`?
- What about `add()`?