L11: Linked List and Pointer Problems

Alex Steiger

CompSci 201: Spring 2024

2/19/2024

Logistics, Coming up

- Monday, 2/19 (today)
  - Project 2: Markov Due
  - Project 3: DNA out by tomorrow
- Thursday, 2/22
  - APT Quiz 1 due
- Next Monday 2/26
  - Nothing due ☺
  - Work on Project 3: DNA, due the following week

Today's Outline

- Part 1: LinkedList review + low-level details
- Part 2: Implementing DIYLinkedList
- Part 3 / Wed: Working directly with List Node objects, algorithmic problem-solving
  1. Get to index'th node
  2. Append one list to another
  3. Reverse a list in place

What is the runtime complexity of the reverseCopy method as a function of n where n is the size of myList? *

```
22    public static List<Integer> reverseCopy(LinkedList<Integer> myList) {
23        List<Integer> reversed = new LinkedList<>();
24        for (Integer val : myList) {
25            // adds val to front of list
26            reversed.add(0, val);
27        }
28        return reversed;
29    }
```

- ◯ O(1)
- ✓ O(n)
- ◯ O(n^2)
- ◯ O(n^3)

3

What is the runtime complexity of the removeZeros method be as a function of n, the number of elements in the list? Answer in the worst case / without making any assumptions about the elements of the input myList. *

```
 8    public static void removeZeros(LinkedList<Integer> myList) {
 9        for (int i=0; i<myList.size(); i++) {
10            if (myList.get(i) == 0) {
11                myList.remove(i);
12            }
13        }
```

- ◯ O(1)
- ◯ O(n)
- ✓ O(n^2)
- ◯ O(n^3)

4

What is the runtime complexity of the removeZeros method be as a function of n, the number of elements in the list? Answer in the worst case / without making any assumptions about the elements of the input myList.

The Java API documentation clarifies that the remove() method on an Iterator "Removes from the underlying collection the last element returned by this iterator." *

```
6       public static void removeZeros(LinkedList<Integer> myList) {
7           Iterator<Integer> listIter = myList.iterator();
8           while (listIter.hasNext()) {
9               if (listIter.next() == 0) {
10                  listIter.remove();
11              }
12          }
13      }
```

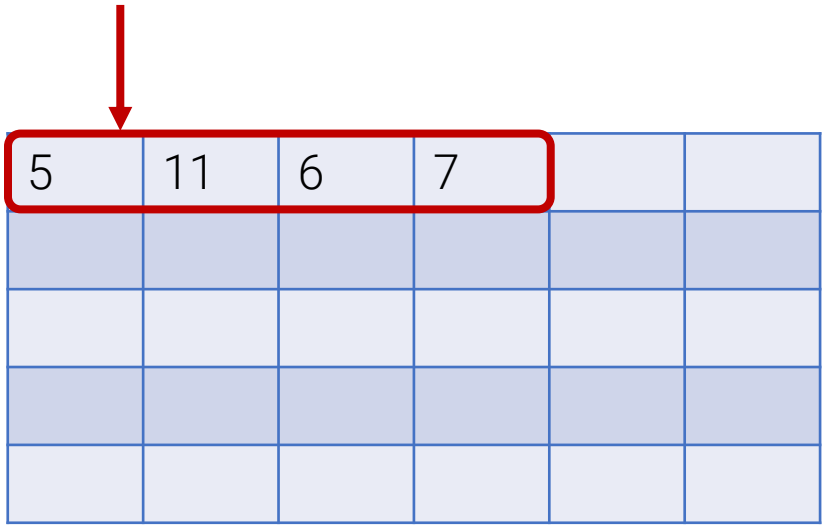
- ◯ O(1)
- ✓ O(n)
- ◯ O(n^2)
- ◯ O(n^3)

Linked List, Low-level DIY perspective

Contrasting how things look to your computer / in memory

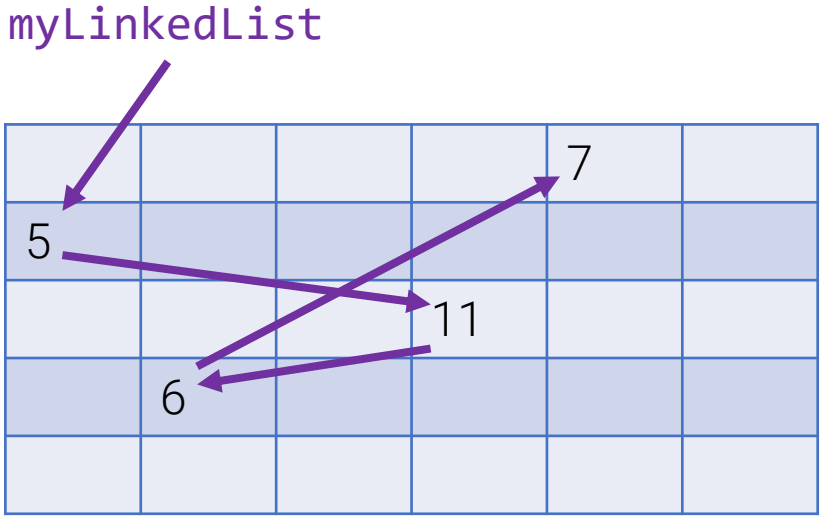
Array/ArrayList

Elements laid out sequentially, one at a time, in order, in memory.



LinkedList

Elements at *arbitrary* locations in memory, connected only by references to the next element.

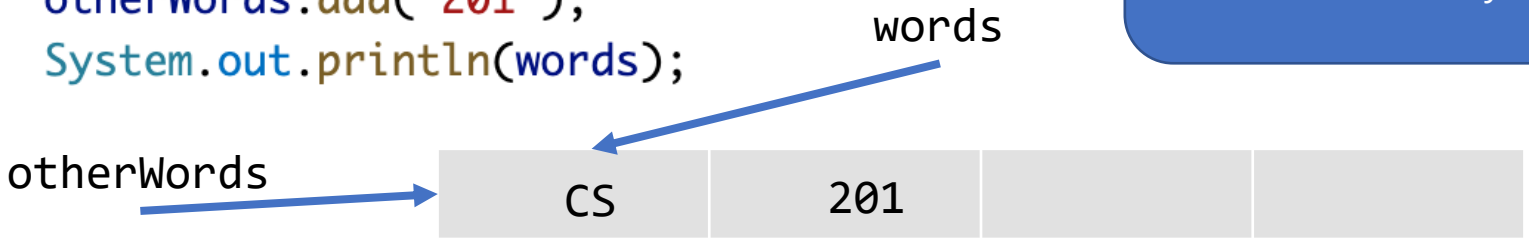


Memory and references

- In Java, **variables for *reference types*** (anything that is an object/not a primitive) really **store the location of the object in memory**.
- Can have *multiple references* to the same object in memory!

```
6       List<String> words = new LinkedList<>();
7       words.add("CS");
8       List<String> otherWords = words;
9       otherWords.add("201");
10      System.out.println(words);
```

Prints [“CS”, “201”], only one actual List in memory!



Multiple objects or multiple references

Java creates a reference type object in memory only when the code calls the **new** operator.

```
11      List<String> listA = new LinkedList<>();
12      List<String> listB = new LinkedList<>();
```

First example create 2 *distinct* empty lists, but…

```
11      List<String> listA = new LinkedList<>();
12      List<String> listB = listA;
```

Second example creates *one* list in memory with two references / variable names.

Pass by value of reference

```
12    public static void removeFront(List<String> words) {
13        words.remove(0);
14    }
```

- Java does NOT copy all of **`words`** when we call this method.
- Copies the *reference* (memory address) and passes that, O(1) time [memory addresses are 64 bits].
- Changes relevant outside of method.

```
6     List<String> words = new LinkedList<>();
7     words.add("CS");
8     removeFront(words);
9     System.out.println(words);
```

Prints [] (empty), change to words in method changes the only List in memory. Different for primitive types.

More Pass by value of reference

- Why does it matter that Java passes a *copy* of the reference to methods?
- Cannot "lose" a reference inside a method.

```
16    public static void tryBreakReference(List<String> words) {
17        words = new LinkedList<>();
18    }
```

Even though this reassigns **words** in the method…

```
6     List<String> words = new LinkedList<>();
7     words.add("CS");
8     tryBreakReference(words);
9     System.out.println(words);
```

Still prints ["CS"], only the *copy* of the reference was reassigned.

Null reference/pointer

- The default value for an uninitialized (no memory allocated by a call to new) object is **`null.`**

- Can check if an object `== null`.
  - We will use to denote the end of a linked list, the node with no more nodes following.

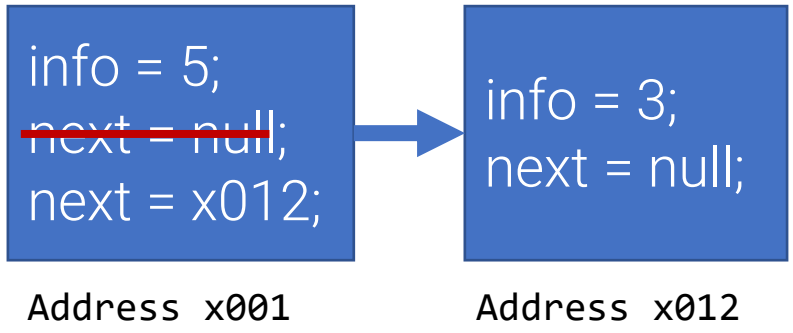
- If you try to call any methods on a null object, will get a **`NullPointerException`** error.

Linked list is a list implemented by linked nodes. What is a node?

- Just a Java object of a class we write, like any other!
- We want to "link" them together, so each node has a *pointer* (really a reference = a memory location) to another node.

```
public class ListNode {
    int info;
    ListNode next;
    ListNode(int x){
        info = x;
    }
    ListNode(int x, ListNode node){
        info = x;
        next = node;
    }
}
```

```
ListNode first = new ListNode(5);
ListNode second = new ListNode(3);
first.next = second;
```



Creating and traversing a linked list

- **`ListNode`** class used in APTs, etc.
  - The variable for the “linked list itself” is just a reference to the first **`ListNode`**


```
ListNode list = new ListNode(5);
list.next = new ListNode(7);
list.next.next = new ListNode(9);
print(list);
```

```
public static void printList(ListNode list) {
    while(list != null) {
        System.out.println(list.info);
        list = list.next;
    }
}
```

While there is a next node…

Print value of current node

Go to next node

Creating Nodes, constructing lists

1. Calling `new Node(…)` always creates a Node in memory that did not exist before

2. Writing `node.next = otherNode;` makes node → (point to) otherNode

3. `node.next` or `node.info` gives an error (null pointer exception) if node is null

2

This and following questions reference the ListNode class shown. Suppose we run the following code:

ListNode myList =new ListNode(2, new ListNode(0, new ListNode(1)));

What is myList.next.next? *

- ○ 0
- ○ The second ListNode object
- ○ 1
- ✓ The third ListNode object
- ○ null

```
1   public class ListNode {
2       int info;
3       ListNode next;
4       public ListNode(int info) {
5           this.info = info;
6       }
7       public ListNode(int info, ListNode next) {
8           this.info = info;
9           this.next = next;
10      }
11  }
```

3

Again suppose we run the following code.

ListNode myList =new ListNode(2, new ListNode(0, new ListNode(1)));

What is myList.next.info? *

- [x] 0
- [ ] The second ListNode object
- [ ] 1
- [ ] The third ListNode object
- [ ] null

```
1   public class ListNode {
2       int info;
3       ListNode next;
4       public ListNode(int info) {
5           this.info = info;
6       }
7       public ListNode(int info, ListNode next) {
8           this.info = info;
9           this.next = next;
10      }
11  }
```

4

Again suppose we run the following code.

ListNode myList =new ListNode(2, new ListNode(0, new ListNode(1)));

What is myList.next.next.next? *

- ◯ 1
- ◯ The third ListNode object
- ☑ null
- ◯ error, null pointer exception

```
1   public class ListNode {
2       int info;
3       ListNode next;
4       public ListNode(int info) {
5           this.info = info;
6       }
7       public ListNode(int info, ListNode next) {
8           this.info = info;
9           this.next = next;
10      }
11  }
```

myList.next.next.next.next causes a NullPointerException

5

Consider the following code. assume the printList method prints the values in a list (meaning everything from a given starting ListNode and following next references until reaching null). What would be printed by **line 18**, which prints **ret**? *

```
 9      public static ListNode foo(ListNode list) {
10          list = list.next;
11          list.next = null;
12          return list;
13      }
14
        Run | Debug
15      public static void main(String[] args) {
16          ListNode list = new ListNode(info: 2, new ListNode(info: 0, new ListNode(info: 1)));
17          ListNode ret = foo(list);
18          printList(ret);
19          printList(list);
20      }
```

- ◯ nothing
- ☑ 0
- ◯ 2, 0
- ◯ 2, 0, 1

Same code. What would be printed by **line 19**, which prints **list**? *

```
 9      public static ListNode foo(ListNode list) {
10          list = list.next;
11          list.next = null;
12          return list;
13      }
14
        Run | Debug
15      public static void main(String[] args) {
16          ListNode list = new ListNode(info: 2, new ListNode(info: 0, new ListNode(info: 1)));
17          ListNode ret = foo(list);
18          printList(ret);
19          printList(list);
20      }
```

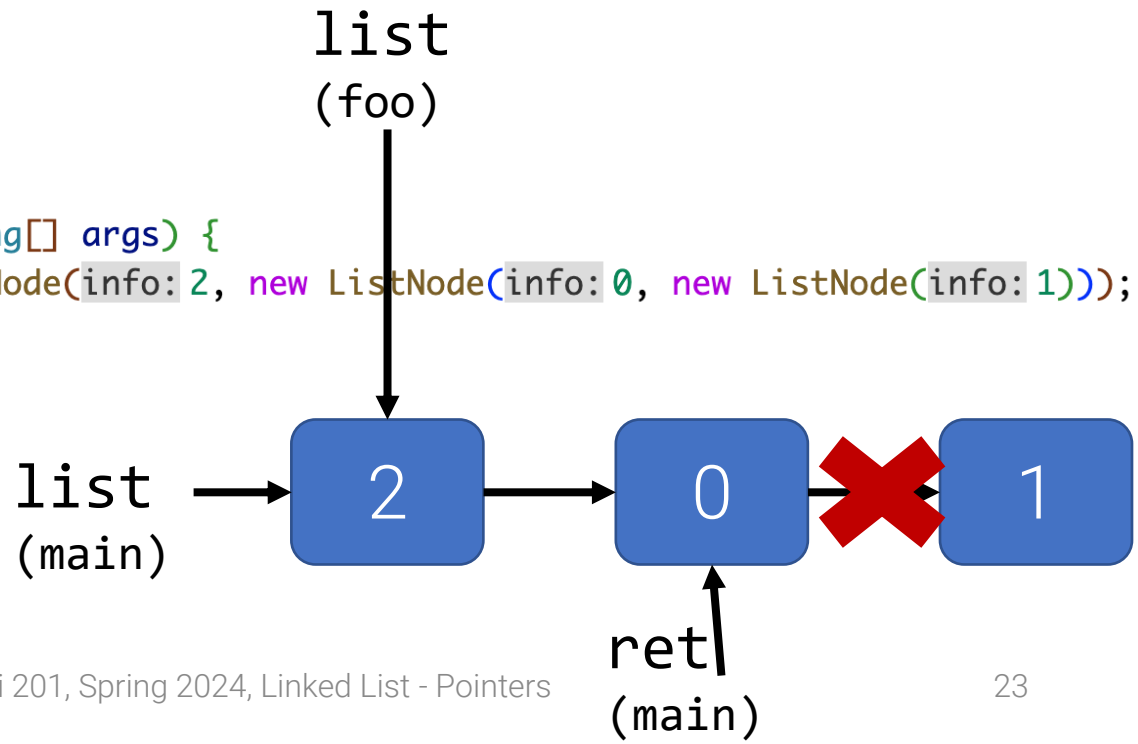
- ◯ nothing
- ◯ 0
- ✓ 2, 0
- ◯ 2, 0, 1

WOTO Answers

What would line 18 print? 0

What would line 19 print? 2, 0

```
 9    public static ListNode foo(ListNode list) {
10        list = list.next;
11        list.next = null;
12        return list;
13    }
14
      Run | Debug
15    public static void main(String[] args) {
16        ListNode list = new ListNode(info: 2, new ListNode(info: 0, new ListNode(info: 1)));
17        ListNode ret = foo(list);
18        printList(ret);
19        printList(list);
20    }
```



DIYLinkedList

Live Coding