L9: Memory, Pointers, LinkedList

Alex Steiger

CompSci 201: Spring 2024

2/14/2024

1

Announcements, Coming up

- Today, Wednesday 2/14
  - APT 4 due
- Next Monday 2/19
  - Project P2: Markov due
- Next Wednesday 2/21
  - APT **Quiz 1** due

2

Summer course book bagging is open – course offerings in CS

Summer Term 1 (May 15 – June 27)

- CS 230 Discrete Math
  - Mathematical notations, logic, and proof; linear and matrix algebra; graphs, digraphs, trees, representations, and algorithms; counting, permutations, combinations, discrete probability, Markov models; advanced topics from algebraic structures, geometric structures, combinatorial optimization, number theory. Pre/corequisite: Computer Science 201.
- CS 250 Intro. Design and Analysis of Algorithms
  - Computer structure, assembly language, instruction execution, addressing techniques, and digital representation of data. Computer system organization, logic design, microprogramming, cache and memory systems, and input/output interfaces. Pre/corequisite: Computer Science 201.

3

Summer course book bagging is open – course offerings in CS

Summer Term 2 (July 1 – August 11)

- CS 330 Intro. Design and Analysis of Algorithms
  - Design and analysis of efficient algorithms including sorting, searching, dynamic programming, graph algorithms, fast multiplication, and others; nondeterministic algorithms and computationally hard problems. Pre/corequisite: Computer Science 201.
- CS 207 Intro. iOS Mobile Programming
  - This class explores the world of mobile applications development based on Apple's iOS operating system and Swift programming language. The class will work on Mac computers running Xcode, the integrated development environment, to develop applications for iPhone/iPad devices. The class covers fundamentals essential to understanding all aspects of app development from concept to deployment on the App Store. Students required to present their project proposals and deliver a fully functional mobile application as a final project.

4

What is an APT Quiz?

- Set of 3 APT problems, 2 hours to complete.
  - Will be available starting this Saturday afternoon (look for a Canvas/email announcement)
  - Must *complete* by 11:59 pm Wednesday 10/18 (so start before 10pm)
- Start the quiz from Instructions Doc on Canvas: shows you the link to the problems and submission page; clicking link ***begins your timer.***
  - Will look/work just like the regular APT page, just with only 3 problems.

5

What is allowed?

Yes, allowed

- zyBook
- Course notes
- API documentation
- VS Code
- JShell

No, not allowed

- Collaboration or sharing any code.
- Communication about the problems ***at all*** during the window.
- Searching internet, stackoverflow, etc. for solutions.

6

Don't do these things

1. Do not collaborate. Note that we log all code submissions and will investigate for academic integrity.

2. Do not hard code the test cases (if(input == X) return Y, etc.).
   We show you the test cases to help you debug. But we search for submissions that do this and **you will get a 0 on the APT quiz if you hard code the test cases** instead of solving the problem.

7

Do:

- Make a ***Cloud Recording*** on Zoom
  - Start ***before*** you click link in instruction doc
  - Submit URL via Form (like P0, P1)

- ***Must*** be a Cloud Recording!
  - Penalty for missing/broken Zoom URL

8

How is it graded?

Not curved; adjusted. 3 problems, 10 points each.

Raw score R out of 30.	Adjusted score A out of 30.	100 point grade scale
27 <= R <= 30	A = R	90 – 100
24 <= R <= 26	A = 26	~87
21 <= R <= 23	A = 25	~83
18 <= R <= 20	A = 24	80
15 <= R <= 17	A = 23	~77
12 <= R <= 14	A = 22	~73
9 <= R <= 11	A = 21	70
6 <= R <= 8	A = 20	~67
3 <= R <= 5	A = 19	~63
1 <= R <= 2	A = 18	60

Can still get in the B range even if you can't solve one; don't panic!

Only going to get a 0 if you collaborate or hard code test cases. Don't do it!

9

Some Exam 1 Problems

10

Big O: Composition

11

Runtime complexity of composed methods

- Runtime complexity of **`stuff(stuff(n))`**?

```
7   public int stuff(int n) {
8       int sum = 0;
9       for(int k=0; k < n; k += 1) {
10          sum += n;
11      }
12      return sum;
13  }
```

- Value returned by **`stuff(n)`** is n^2.
- Runtime complexity of **`stuff(n²)`**?

- **`stuff`** has linear runtime complexity, so **`stuff(n²)`** is $O(n^2)$

12

Composing methods general

- Given two methods:

```
public static int outer (int n) {
public static int inner(int n) {
```

- What is the runtime complexity of the following?

```
int result = outer(inner(n));
```

Running this code is equivalent to…

```
int innerValue = inner(n);
int result = outer(innerValue);
```


13

Composing methods general

- Given two methods:

```
public static int outer (int n) {
public static int inner(int n) {
```

- What is the runtime complexity of the following?

```
int result = outer(inner(n));
```

Three steps: Runtime complexity is Step1+Step3.

1. Calculate runtime complexity of `inner(n)`
2. Calculate value returned by `inner(n)`
3. Calculate runtime complexity of `outer()` on value from step 2.

14

Composing methods example

```
int result = outer(inner(n));
```

```
56  public static int outer (int n) {
57      int result = 0;
58      for (int i=1; i<n; i*=2) {
59          result += 1;
60      }
61      return result;
62  }
63
64  public static int inner(int n) {
65      return n*n;
66  }
```

1. Runtime complexity of `inner(n)` is O(1)
2. Value returned by `inner(n)` is $O(n^2)$
3. Runtime complexity of outer(n^2) is $O(\log(n^2))$

Recall log rules: $\log(n^2) = 2\log(n)$

Total runtime complexity: $O(1) + O(\log(n^2))$ is $O(\log(n))$
Most of the "work" done executing `outer`

15

Another composition example

```
int result = outer(inner(n));
```

```
56  public static int outer (int n) {
57      int result = 0;
58      for (int i=1; i<n; i*=2) {
59          result += 1;
60      }
61      return result;
62  }
63
64  public static int inner(int n) {
65      int result = 0;
66      for (int i=0; i<n; i++) {
67          result += n;
68      }
69      return result;
70  }
```

1. Runtime complexity of `inner(n)` is now O(n)
2. Value returned by `inner(n)` is still $O(n^2)$
3. Runtime complexity of outer(n^2) is still $O(\log(n^2))$

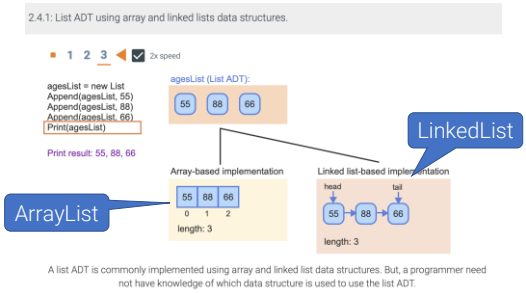
Total runtime complexity: O(n) + $O(\log(n^2))$ is O(n)
Now most of the "work" done executing `inner`

16

Linked List, API Perspective

17

Multiple Implementations of the Same Interface



18

Motivating List Interface Implementations by Efficiency

- `List<String> a = new LinkedList<>();`
- `List<String> b = new ArrayList<>();`

You already know how to use a List, same exact methods and functionality with LinkedList!

- Implementation? ArrayList implements List using Array, LinkedList implements List using…"links"?
- Tradeoffs? Which is more efficient (for ___)?

19

ArrayList uses Array. Fast random access memory, fast get()

- Accessing Array (or ArrayList `get(i)`) at index i takes the same time whether:
  - i=1, 201, 2001, …
- Possible because Java compiler knows:
  - Where in memory the array starts (say position X),
  - array is laid out consecutively, all together, in memory,
  - Memory each value takes (say 4 bytes per int).
- Allows to calculate the memory position of `myArray[i]` in constant time (more in CS 210/250).

20

Pros/Cons of Array-Based Data Structures

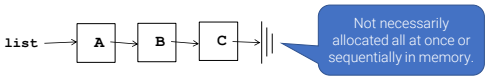
Array-Based Data Structure	What array?
`ArrayList`	Array of list elements
`String/StringBuilder`	Array of characters
`HashSet/Map`	Array of buckets

Pros	Cons
O(1) lookup by index	Hard to add/remove except at the end.
Little memory overhead, just storing elements	Adding elements gives amortized (averaged) efficiency, not worst case.

21

What is a (singly) linked list conceptually?

A reference (~pointer) to the *first* node in a list, connected by a reference (~pointer) to the next node.

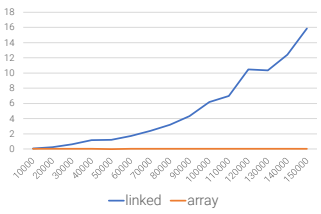


No constant time access to nodes in the middle. To get C, start at A, follow the references (~pointers).

22

ArrayList much faster than LinkedList for Random Access `.get()` operations

list size	linked	array
10000	0.0583	0.0012
20000	0.223	0.0014
30000	0.6	0.0009
40000	1.1643	0.0008
50000	1.1847	0.0007
60000	1.703	0.001
70000	2.3685	0.0013
80000	3.1883	0.0015
90000	4.3096	0.0017
100000	6.1647	0.0021
110000	6.9777	0.0038
120000	10.4757	0.0026
130000	10.3337	0.003
140000	12.4032	0.0032
150000	15.8398	0.0059



23

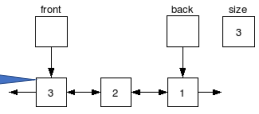
LinkedList `.get()` runtime explained

- Calling **`list.get(k)`** is O(N) for LinkedList
  - Not quite, O(min(*k, size-k*), doubly-linked list
  - **`list.get(k)`** is O(1) for ArrayList

Worst-case is still O(N)

- To get every element one at a time:
  - Linked: $2(1 + 2 + \ldots + N/2)$ is $O(N^2)$
  - Array: $1 + 1 + \ldots + 1$ is O(N)

Java API LinkedList is actually doubly-linked, pointers forward and back.



24

`get()` vs. Iterator

For LinkedList `lList` of N integers…

```
17  // Looping with get
18  for (int i=0; i<N; i++) {
19      total += lList.get(i);
20  }
21
22  // Looping with iterator (implicit)
23  for (int val : lList) {
24      total += val;
25  }
26
27  // Looping with iterator (explicit)
28  Iterator<Integer> listIter = lList.iterator();
29  while (listIter.hasNext()) {
30      total += listIter.next();
31  }
```

This loop is $O(N^2)$

N	Runtime in s Using get	Runtime in s with Iterator
25k	0.2	0.0 (rounding)
50k	0.9	0.0 (rounding)
100k	3.9	0.0 (rounding)
200k	16.2	0.0 (rounding)

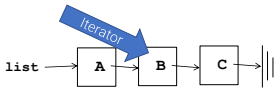
Also O(N)

Equivalent to second loop, hasNext and next just like Scanner

25

What is an Iterator conceptually?

- get() method always starts at the front of the list.
- Iterator maintains current position in list.



Looping with get()
get(i) → Start at beginning, iterate over i-1 elements.

Looping with iterator
Next element where iterator is pointing, then advance iterator.

26

Are LinkedLists just worse? Removing from the front

For LinkedList `lList` and ArrayList `aList` of N integers…

```
double before = System.nanoTime();
for (int t=0; t<n; t++) {
    lList.remove( index: 0);
}
double after = System.nanoTime();
System.out.println((after-before)/1e9);
```

VS

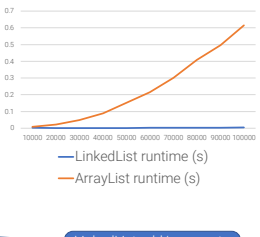
```
before = System.nanoTime();
for (int t=0; t<n; t++) {
    aList.remove( index: 0);
}
after = System.nanoTime();
System.out.println((after-before)/1e9);
```

Timing repeatedly removing from the front…

27

LinkedList remove/add to front empirical results

List Size	LinkedList runtime (s)	ArrayList runtime (s)
10000	0.002	0.008
20000	0.001	0.022
30000	0.001	0.049
40000	0.001	0.088
50000	0.001	0.152
60000	0.002	0.216
70000	0.003	0.301
80000	0.003	0.409
90000	0.003	0.497
100000	0.004	0.615

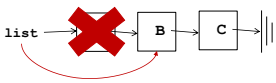


LinkedList add/remove to front are O(1) (so remove N from front is O(N)

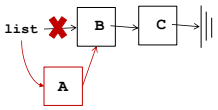
28

Explaining fast remove/add to front for LinkedList

To remove from the front, Just update list to point to the second element. No other shifting!



To add to the front, just make a new node pointing to the second element. No shifting!



29

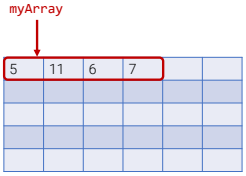
Linked List, Low-level DIY perspective

34

Contrasting how things look to your computer / in memory

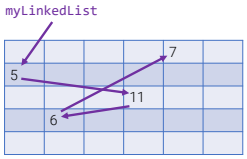
Array/ArrayList

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.



35

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!

36

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.

37

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.

38

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.

39

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 **null pointer exception error**.

40

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;
```

41

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

42

DIYLinkedList

Live Coding 

49