L9: Memory, Pointers, LinkedList

Alex Steiger
CompSci 201: Spring 2024
2/14/2024
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
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.
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.
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.
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.
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.
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
How is it graded?

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

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

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!
Some Exam 1 Problems
Big O: Composition
Runtime complexity of composed methods

• Runtime complexity of \texttt{stuff(stuff(n))}?

```java
public int stuff(int n) {
    int sum = 0;
    for(int k=0; k < n; k += 1) {
        sum += n;
    }
    return sum;
}
```

• Value returned by \texttt{stuff(n)} is \(n^2\).

• Runtime complexity of \texttt{stuff(n^2)}?

• \texttt{stuff} has linear runtime complexity, so \texttt{stuff(n^2)} is \(O(n^2)\)
Composing methods general

• Given two methods:

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

• What is the runtime complexity of the following?

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

Running this code is equivalent to...

```java
int innerValue = inner(n);
int result = outer(innerValue);
```
Composing methods general

• Given two methods:

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

• What is the runtime complexity of the following?

```java
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.
**Composing methods example**

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

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 $O(\log(n^2))$

Total runtime complexity: $O(1) + O(\log(n^2))$ is $O(\log(n))$

Most of the “work” done executing `outer`

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

```java
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`
Linked List, API Perspective
Multiple Implementations of the Same Interface

2.4.1: List ADT using array and linked lists data structures.

A list ADT is commonly implemented using array and linked list data structures. But, a programmer need not have knowledge of which data structure is used to use the list ADT.
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 ___)?
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).
Pros/Cons of Array-Based Data Structures

<table>
<thead>
<tr>
<th>Array-Based Data Structure</th>
<th>What array?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>Array of list elements</td>
</tr>
<tr>
<td>String/StringBuilder</td>
<td>Array of characters</td>
</tr>
<tr>
<td>HashSet/Map</td>
<td>Array of buckets</td>
</tr>
</tbody>
</table>

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

Not necessarily allocated all at once or sequentially in memory.
ArrayList much faster than LinkedList for Random Access `get()` operations

<table>
<thead>
<tr>
<th>list size</th>
<th>linked</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.0583</td>
<td>0.0012</td>
</tr>
<tr>
<td>20000</td>
<td>0.223</td>
<td>0.0014</td>
</tr>
<tr>
<td>30000</td>
<td>0.6</td>
<td>0.0009</td>
</tr>
<tr>
<td>40000</td>
<td>1.1643</td>
<td>0.0008</td>
</tr>
<tr>
<td>50000</td>
<td>1.1847</td>
<td>0.0007</td>
</tr>
<tr>
<td>60000</td>
<td>1.703</td>
<td>0.001</td>
</tr>
<tr>
<td>70000</td>
<td>2.3685</td>
<td>0.0013</td>
</tr>
<tr>
<td>80000</td>
<td>3.1883</td>
<td>0.0015</td>
</tr>
<tr>
<td>90000</td>
<td>4.3096</td>
<td>0.0017</td>
</tr>
<tr>
<td>100000</td>
<td>6.1647</td>
<td>0.0021</td>
</tr>
<tr>
<td>110000</td>
<td>6.9777</td>
<td>0.0038</td>
</tr>
<tr>
<td>120000</td>
<td>10.4757</td>
<td>0.0026</td>
</tr>
<tr>
<td>130000</td>
<td>10.3337</td>
<td>0.003</td>
</tr>
<tr>
<td>140000</td>
<td>12.4032</td>
<td>0.0032</td>
</tr>
<tr>
<td>150000</td>
<td>15.8398</td>
<td>0.0059</td>
</tr>
</tbody>
</table>
LinkedList\texttt{.get()} runtime explained

- Calling \texttt{list.get(k)} is $O(N)$ for LinkedList
  - Not quite, $O(\min(k, \text{size}-k))$, doubly-linked list
  - \texttt{list.get(k)} is $O(1)$ for ArrayList
- 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.

Worst-case is still $O(N)$
get() vs. Iterator

For LinkedList `lList` of N integers...

```java
    // Looping with get
    for (int i=0; i<N; i++) {
        total += lList.get(i);
    }

    // Looping with iterator (implicit)
    for (int val : lList) {
        total += val;
    }

    // Looping with iterator (explicit)
    Iterator<Integer> listIter = lList.iterator();
    while (listIter.hasNext()) {
        total += listIter.next();
    }
```

**Runtime Comparison**

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime in s Using get</th>
<th>Runtime in s with Iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>25k</td>
<td>0.2</td>
<td>0.0 (rounding)</td>
</tr>
<tr>
<td>50k</td>
<td>0.9</td>
<td>0.0 (rounding)</td>
</tr>
<tr>
<td>100k</td>
<td>3.9</td>
<td>0.0 (rounding)</td>
</tr>
<tr>
<td>200k</td>
<td>16.2</td>
<td>0.0 (rounding)</td>
</tr>
</tbody>
</table>

This loop is \(O(N^2)\)

Also \(O(N)\)

Equivalent to second loop, hasNext and next just like Scanner
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.
Are LinkedLists just worse? Removing from the front

For LinkedList **lList** and ArrayList **aList** of N integers...

```java
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);
```

```java
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...
LinkedList remove/add to front empirical results

<table>
<thead>
<tr>
<th>List Size</th>
<th>LinkedList runtime (s)</th>
<th>ArrayList runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.002</td>
<td>0.008</td>
</tr>
<tr>
<td>20000</td>
<td>0.001</td>
<td>0.022</td>
</tr>
<tr>
<td>30000</td>
<td>0.001</td>
<td>0.049</td>
</tr>
<tr>
<td>40000</td>
<td>0.001</td>
<td>0.088</td>
</tr>
<tr>
<td>50000</td>
<td>0.001</td>
<td>0.152</td>
</tr>
<tr>
<td>60000</td>
<td>0.002</td>
<td>0.216</td>
</tr>
<tr>
<td>70000</td>
<td>0.003</td>
<td>0.301</td>
</tr>
<tr>
<td>80000</td>
<td>0.003</td>
<td>0.409</td>
</tr>
<tr>
<td>90000</td>
<td>0.003</td>
<td>0.497</td>
</tr>
<tr>
<td>100000</td>
<td>0.004</td>
<td>0.615</td>
</tr>
</tbody>
</table>

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

myArray

| 5 | 11 | 6 | 7 |

LinkedList

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

myLinkedList

| 5 | 6 | 11 | 7 |
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!

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

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

```java
define
    List<String> listA = new LinkedList<>();
define
    List<String> listB = new LinkedList<>();
```

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

```java
define
    List<String> listA = new LinkedList<>();
define
    List<String> listB = listA;
```

Second example creates one list in memory with two references / variable names.
Pass by value of reference

- 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.

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

```java
List<String> words = new LinkedList<>();
words.add("CS");
removeFront(words);
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.

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

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

Even though this reassigns `words` in the method...

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 `null pointer exception` 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.

```java
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;
```

```
info = 5;
next = null;
next = x012;
```

```
Address x001
```

```
info = 3;
next = null;
```

```
Address x012
```
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
DIYLinkedList

Live Coding