CompSci 201, L9: Memory, Pointers, LinkedList
Logistics, Coming up

• Today, Monday 9/26
  • Intro LinkedList (not on exam)

• Wednesday 9/28
  • Exam 1 in class
    • 60 minutes long, try to arrive on time or early to get set up
    • Can bring a 1 page reference/study sheet
  • APT 4 due

• Monday 10/3
  • Project 2: Markov Due
  • More LinkedList
Person in CS: Alan Turing

• 1912-1954 (died at 41)
• English, PhD at Princeton in 1938
• Mathematician, cryptographer, pioneering thinker in AI
  • “Father of modern computer science”
  • Turing machine – helped formalize what is computable
  • Cryptography work in WW2
• Prosecuted in 1952 for homosexuality
  • Given choice of chemical “treatment” or prison, took former
  • Died 2 years later of cyanide poisoning, circumstances debated
Outline for Today

1. Review Big O

2. LinkedList: API Perspective

3. LinkedList: DIY Low-level Perspective

(Next time – LinkedList low-level and problem solving)
Reviewing Big O
Runtime complexity of composed methods

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

\begin{verbatim}
public int stuff(int n) {
    int sum = 0;
    for(int k=0; k < n; k += 1) {
        sum += n;
    }
    return sum;
}
\end{verbatim}

• 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?

    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 1+3.
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`
Another composition example

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

public static int outer (int n) {
    int result = 0;
    for (int i=1; i<n; i*=2) {
        result += 1;
    }
    return result;
}

public static int inner(int n) {
    int result = 0;
    for (int i=0; i<n; i++) {
        result += n;
    }
    return result;
}
```

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`
WOTO

Go to duke.is/8fmw6

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!
Linked List, API Perspective
Multiple Implementations of the Same Interface

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

```
agesList = new List
Append(agesList, 55)
Append(agesList, 88)
Append(agesList, 66)
Print(agesList)
```

Print result: 55, 88, 66

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

• Calling \texttt{list.get(k)} is O(N) for LinkedList
  • Not quite, O(\text{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.
get() vs. Iterator

For LinkedList lList of N integers...

```java
17    // Looping with get
18    for (int i=0; i<N; i++) {
19        total += lList.get(i);
20    }

22    // Looping with iterator (implicit)
23    for (int val : lList) {
24        total += val;
25    }

27    // Looping with iterator (explicit)
28    Iterator<Integer> listIter = lList.iterator();
29    while (listIter.hasNext()) {
30        total += listIter.next();
31    }
```

<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 $O(N^2)$

These loops are $O(N)$

Equivalent to second loop, hasNext and next just like Scanner
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...
Linked List 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>

Linked List add/remove to front are $O(1)$ (so remove $N$ from front is $O(N)$)
WOTO

Go to duke.is/cy43f

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!
Linked List, Low-level DIY perspective
Low-level DIY perspective on linked lists: why?

Less important reasons?
• Tradition, “everyone else did”, etc.
• Will be useful in later courses (e.g., CS 210/250), not all languages have a List built-in (e.g., C)

More important reasons?
• Deepen understanding of how objects are represented in memory, references and pointers.
• Foundation for dynamic/linked data structures (example: trees).
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);
```

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<>();
```

Creates two distinct empty lists in memory, but

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

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
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;
    }
}
```

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

```
info = 5;
next = null;
info = 3;
next = x012;
```

Address x001
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