CIS121-204–Fall 2007

Lab 7 Solution

Tuesday, October 30

Appending Two Lists

Recall that **s1** has size **m** and **s2** has size **n**. We assume that **ArrayList** is never full. Otherwise, we can use amortized analysis to derive the following running time; instead of the worst-case running time it will be amortized running time.

Approach 1

```
01 public static <E> List<E> append1(List<E> s1, List<E> s2)
02 {
03   List<E> l = new ArrayList(); // or new LinkedList();
04
05   for (int i=0; i<s1.size(); i++)
06     l.add(l.size(), s1.get(i)); // copy s1 into l
07
08   for (int i=0; i<s2.size(); i++)
09     l.add(l.size(), s2.get(i)); // copy s2 into l
10
11   return l;
12 }
```

First we analyze line 6. There are three operations done in this line:

- **l.size()**: [1] This operation takes $O(1)$ in all implementations.
- **s1.get(i)**: [2] This operation takes $O(1)$ in **ArrayList** implementation and $O(i)$ in **LinkedList** implementation
- **l.add(l.size(),o)**: [3] This operation takes $O(1)$ in **ArrayList** implementation, $O(1)$ in **LinkedList** implementation with a link to the last element, and $O(i)$ in **LinkedList** implementation without the link.

So, the **for** loop in lines 5-6 runs m times, each time it takes [1] + [2] + [3], total of $O(m)$ in **ArrayList**, $O(m^2)$ in **LinkedList** with the link, and $O(m^2)$ in **LinkedList** without the link.

Now we analyze line 9. First of all note that **l** has size **m** before entering the loop in line 8. There are three operations done in this line:

- **l.size()**: [1] This operation takes $O(1)$ in all implementations.
- **s2.get(i)**: [2] This operation takes $O(1)$ in **ArrayList** implementation and $O(i)$ in **LinkedList** implementation
- **l.add(l.size(),o)**: [3] This operation takes $O(1)$ in **ArrayList** implementation, $O(1)$ in **LinkedList** implementation with a link to the last element, and $O(m+i)$ in **LinkedList** implementation without the link.

So, the **for** loop in lines 8-9 runs n times, each time it takes [1] + [2] + [3], total of $O(n)$ in **ArrayList**, $O(n^2)$ in **LinkedList** with the link, and $O(mn+n^2)$ in **LinkedList** without the link.

Summarizing,

- **append1()** runs in $O(m+n)$ in **ArrayList** implementation.
- **append1()** runs in $O(m^2+n^2)$ in **LinkedList** implementation with a link to the last element.
- **append1()** runs in $O(m^2+mn+n^2) = O(m^2+n^2)$ (why?) in **LinkedList** implementation without a link to the last element.

Approach 2

```
01 public static <E> List<E> append2(List<E> s1, List<E> s2)
02 {
03   if (s2.size() == 0) // test if second list is empty
04     return s1;
05   else {
06     E o = s2.remove(s2.size()-1); // last of s2
07     List<E> l = append2(s1, s2); // recursive call with smaller s2
08     l.add(l.size(),o); // last of s2 is added after the recursive call
09     return l;
10   }
11 }
```

Let $T(i, j)$ be the time to append **s1** of size i and **s2** of size j. We want to calculate $T(m, n)$. First of all, note that $T(i, 0) = O(1)$ for all i. Otherwise, we have

$$T(i, j) = [6] + T(i, j-1) + [8],$$

where

- [6] is the running time of line 6, which is $O(1)$ for **ArrayList**, $O(1)$ for **LinkedList** with a link to the last element, and $O(j)$ for **LinkedList** without the link.
- [8] is the running time of line 8, which is $O(1)$ for **ArrayList**, $O(1)$ for **LinkedList** with a link to the last element, and $O(i+j-1) = O(i+j)$ for **LinkedList** without the link.

Hence,

- For **ArrayList** implementation,

$$\begin{aligned} T(m,n) &= T(m,n-1) + O(1) \\ T(m,n-1) &= T(m,n-2) + O(1) \\ &\vdots \\ T(m,1) &= T(m,0) + O(1) = O(1). \end{aligned}$$

  Hence, $T(m,n) = O(n)$.

- For **LinkedList** implementation with a link to the last element,

$$T(m,n) = T(m,n-1) + O(1).$$

  Hence, $T(m,n) = O(n)$.

- For **LinkedList** implementation without a link to the last element,

$$\begin{aligned}
T(m,n) &= T(m,n-1) + O(n+(m+n)) = T(m,n-1) + O(m+n) \\
T(m,n-1) &= T(m,n-2) + O(m+n-1) \\
T(m,n-2) &= T(m,n-3) + O(m+n-2) \\
&\vdots \\
T(m,1) &= T(m,0) + O(m+1) = O(m+1).
\end{aligned}$$

  Hence, $T(m,n) = O(mn+n^2) = O(m^2+n^2)$ (why?).

Summarizing,

- **append2()** runs in $O(n)$ in **ArrayList** implementation.
- **append2()** runs in $O(n)$ in **LinkedList** implementation with a link to the last element.
- **append2()** runs in $O(m^2+n^2)$ in **LinkedList** implementation without a link to the last element.

Approach 3

```
01 public static <E> List<E> append3(List<E> s1, List<E> s2)
02 {
03   if (s2.size() == 0) // test if second list is empty
04     return s1;
05   else {
06     s1.add(s1.size(),s2.remove(0));
07     return append3(s1, s2); // recursive call with smaller s2
08   }
09 }
```

Let $T(i,j)$ be the time to append **s1** of size i and **s2** of size j. We want to calculate $T(m,n)$. First of all, note that $T(i,0) = O(1)$ for all i. Otherwise, we have

$$T(i,j) = [6] + T(i,j-1),$$

where [6] is the running time of line 6, which contain three operations:

- **s1.size()**: [1] This operation takes $O(1)$ in all implementations.
- **s2.remove(0)**: [2] This operation takes $O(j)$ in **ArrayList** implementation and $O(1)$ in**LinkedList** implementation

- **s1.add(s1.size(),o)**: [3] This operation takes $O(1)$ in **ArrayList** implementation, $O(1)$ in **LinkedList** implementation with a link to the last element, and $O(i)$ in **LinkedList** implementation without the link.

Hence, $[6] = [1] + [2] + [3]$. Now,

- For **ArrayList** implementation,

$$\begin{aligned}
T(m,n) &= T(m+1,n-1) + O(n) \\
T(m+1,n-1) &= T(m+2,n-2) + O(n-1) \\
&\vdots \\
T(m+n-1,1) &= T(m+n,0) + O(1) = O(1).
\end{aligned}$$

  Hence, $T(m,n) = O(n^2)$.

- For **LinkedList** implementation with a link to the last element,

$$\begin{aligned}
T(m,n) &= T(m+1,n-1) + O(1) \\
T(m+1,n-1) &= T(m+2,n-2) + O(1) \\
&\vdots \\
T(m+n-1,1) &= T(m+n,0) + O(1) = O(1).
\end{aligned}$$

  Hence, $T(m,n) = O(n)$.

- For **LinkedList** implementation without a link to the last element,

$$\begin{aligned}
T(m,n) &= T(m+1,n-1) + O(m) \\
T(m+1,n-1) &= T(m+2,n-2) + O(m+1) \\
T(m+2,n-2) &= T(m+3,n-3) + O(m+2) \\
&\vdots \\
T(m+n-1,1) &= T(m+n,0) + O(m+n+1) = O(m+n-1).
\end{aligned}$$

  Hence, $T(m,n) = O(mn+n^2) = O(m^2+n^2)$ (why?).

Summarizing,

- **append3()** runs in $O(n^2)$ in **ArrayList** implementation.
- **append3()** runs in $O(n)$ in **LinkedList** implementation with a link to the last element.
- **append3()** runs in $O(m^2+n^2)$ in **LinkedList** implementation without a link to the last element.

Some Things to Note

- For all the three approaches above, if **`s1==s2`**, no approaches give a correct result. Try examining the code and see what went wrong. What are the results of those erroneous executions.
- **`append3()`** can be implemented without recursion. How?
- Try implementing **`append()`** that works for two identical lists.

Implementing a Stack Using Queues

Yes, we can do that, but how? If you have a solution that you would like to discuss, feel free to come talk to me.

Implementing a Queue Using Stacks

Again, yes, but how? Again, feel free to discuss with me if you think you have a solution.