CS 2230
CS II: Data structures

Meeting 18: higher order functions, cursors
Brandon Myers
University of Iowa
Today’s big ideas

• How to define and use higher order functions\(^1\) in Java by using interfaces

• Overcome limitations of the List ADT with iterators (saw this) and cursors (today)

1. A function that uses functions as data
Higher order functions and data structures

Often it is helpful for data structures to have higher order functions (or, if you want, higher order methods)
• apply a function to every element of a list
• use a function to eliminate some elements

HW5 makes heavy use of higher order functions to separate
• Iteration (dealing with elements)
• “business logic” (doing the actual computation on an element)
HoFs over lists

// apply f to all elements in the list
void applyAll(IntegerFunction f);
### Summary of List ADT operations

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList Time Complexity</th>
<th>LinkedList Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(i, data)</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>get(i)</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>first</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>last</td>
<td>(O(1))</td>
<td>(O(1)) if tail pointer</td>
</tr>
<tr>
<td>append</td>
<td>(O(n)) worst case b/c copy; on avg (O(1))</td>
<td>(O(1)) if tail pointer</td>
</tr>
<tr>
<td>size</td>
<td>(O(1)) (b/c [].length)</td>
<td>(O(1)) if you keep a counter</td>
</tr>
<tr>
<td>applyAll</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>
| remove(i)  | • \(O(1)\) remove will make get/size pay \(O(n)\);  
• \(O(n)\) remove will keep get/size as \(O(1)\) | \(O(n)\) |
| find(data) | \(O(n)\) | \(O(n)\) |

Previously, we saw that for LinkedList, if we plan on doing get(0), get(1), get(2)… then an iterator will improve the efficiency because it keeps track of where you are in the List.
Limitations of the List ADT

• If a List is implemented with LinkedList then we’ve seen that iterating using get() is inefficient because get() takes worst-case $O(n)$ time
  • We fixed this problem with Iterators, which allow us to save where we are and return each element in $O(1)$ time

• How about if we are removing or inserting many specific elements?
  • remove and insert take worst case $O(n)$ time, even if all the insertions are nearby each other
  • Using a doubly-linked list so that we enter from head or tail only cuts the average time in half, which is still $O(n)$
  • A solution:
    • **cursors**: use the
public interface PositionalList<T> {
    public int size();
    public Cursor<T> first();
    public Cursor<T> addLast(T data);
    public Cursor<T> before(Cursor<T> c);
    public Cursor<T> after(Cursor<T> c);
    public Cursor<T> addBefore(Cursor<T> c, T data);
    public Cursor<T> addAfter(Cursor<T> c, T data);
    public void remove(Cursor<T> c);
}

Since a List could be implemented different ways (e.g., ArrayList where position indicated by an int, LinkedList where position), we want to abstract the idea of a position: enter, the Cursor interface.

public interface Cursor<T> {
    T getElement();
}
Today’s big ideas

• How to define and use higher order functions\(^1\) in Java by using interfaces

• Overcome limitations of the List ADT with iterators (saw this) and cursors (today)

\(^1\). A function that uses functions as data