A little more on Lists

CSE 373
Data Structures
Readings

• Chapter 6 Sections 6.1 – 6.4
  › Expandable arrays
  › The “position” concept in the node list ADT
  › Review of iterators (see CSE 143)
  › Collections (read it)
Array Lists

• We have seen the main methods already
• In addition `Java.util.ArrayList`
  › `clear()`
  › `toArray()`
  › `indexOf(e)` (1st occurrence)
  › `lastIndexOf(e)` (last occurrence)
  › See links on the Web for “BasicArrayList”
Extendable Arrays

- Weakness of array implementation: maxsize
- If array occupancy << maxsize => waste of memory
- If array occupancy > maxsize => exception (overflow)
  › For this latter condition, a solution is to expand the array at run-time
Expandable Arrays implementation

- Insert an element in array A of maxsize N when there are already N elements in the array
  - Allocate an array B of size 2N
  - Copy $B[i] := A[i]$, $i = 0, 1, \ldots, N-1$
  - Let $A := B$ (we use B as the array supporting the class)
  - Insert the new element in A

- What happens to the old A?
Cost of Expandable Arrays

- The copy operation is $O(n)$
- If we insert and delete anywhere in the array, the copy is not more costly than an insertion or a deletion
- If we use the array as a stack, insert and delete are $O(1)$
- So is expandable very costly?
  - Yes in the worst case sense but no if …
Amortized cost (informal justification)

- When we expand the array from $N$ to $2N$ we use $O(N)$ extra time.
- However, this will allow to do $N$ insertions (for $i = N, N+1, \ldots, 2N-1$) in $O(1)$ time.
- If we count the time for the copy and the $N$ operations it is $O(N) + N \cdot O(1) = O(N)$.
- So, we do $N$ operations in $O(N)$ time. In an amortized way, when looking at the $N$ insertions, the copy operation costs us constant time.
- For a slightly more formal analysis, see your book pp 229-230.
The Node List ADT

• In the same sense that an element in an array is defined by its index, an element in a list is defined by its position

• Given a list and a position the interface should have methods:
  › Set or replace an element, getfirst, get last, addfirst, removefirst, addafter, removeprevious etc...
  › All of these O(1) is the node list is implemented via a doubly linked list
Iterators

• Lists are ordered collections so very often you want to traverse (walk through) the list
• Iterator extends the concept of position by providing means of stepping to the next element
• Implementation
  › hasNext() tests whether there are elements left in the iterator
  › next() returns the next element in the iterator
Copy singly linked list (version 3 in java)

List dupList = new LinkedList();
for(Iterator i = list.iterator();
    i.hasNext(); )
    dupList.addLast(i.next());

• Of course need to implement the iterator and addLast!
• See web for “BasicLinkedList”