CS310 – Advanced Data Structures and Algorithms

JDK Lists

Spring 2021
A List is an ordered sequence of elements: \( a_0, a_1, a_2, \ldots, a_{n-1} \).

This is a doubly-linked list, the kind used in the JDK LinkedList.

Simple Lists are discussed in Section 1.3 of S&W.

- Diagram from there: singly-linked list
- However, S&W do not offer a general-purpose List API or class

JDK Lists are fully encapsulated, so we never deal directly with the references from item to item. They are doubly-linked.

As you may have guessed, a List ISA Collection
Collection Types: The List Interface

// List interface.
public interface List<AnyType> extends Collection<AnyType> {

    // Returns the item at position idx
    AnyType get(int idx);

    // Changes the item at position idx.
    AnyType set(int idx, AnyType newVal);

    // Obtains a ListIterator object used to traverse the collection bi-directionally.
    ListIterator<AnyType> listIterator(int pos);
}

• Don’t forget all those Collection methods every List gets by the fact that a List ISA Collection. Also the Object methods equals and hashCode since a List ISA Object.
ArrayList and LinkedList implement List

• Just as HashMap and TreeMap implement Map, and HashSet and TreeSet implement Set, we have two major implementations of the List interface, ArrayList and LinkedList
  ➢ ArrayList is based on an array, that grows as needed to hold the elements of the List  
  ➢ LinkedList is the expected chain-linked list

• Good news: we don’t need equals/hashCode/compareTo for elements because the List tolerates/welcomes duplicate elements, unlike Set.

• Example
  List<BankAccount> myList = new ArrayList<BankAccount>();
  or
  List<BankAccount> myList = new LinkedList<BankAccount>();

• By using List as the type of the variable, it’s easy to switch implementations when needed.
// ListIterator interface for List interface.
public interface ListIterator<AnyType> extends Iterator<AnyType> {
    // Tests if there are more items in the collection when iterating in reverse.
    boolean hasPrevious();

    // Obtains the previous item in the collection when traversing in reverse.
    AnyType previous();

    // Remove the last item returned by next or previous.
    // Can only be called once after next or previous.
    void remove();
}

Lists have more powerful iterators than non-List Collections. We can iterate up and down the list. We can start from either end by using the pos argument of List.listIterator(pos).
import java.util.ArrayList;
import java.util.ListIterator;

public class TestArrayList3 {
    public static void main(String[] args) {
        ArrayList<String> array = new ArrayList<String>();
        array.add("apple");
        array.add("banana");
        // Loopthrough the collection by using a ListIterator
        // 1. going forward: or use enhanced for loop here
        ListIterator<String> itr = array.listIterator(0);
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
        // 2. going backwards:
        ListIterator<String> itr2 = array.listIterator(array.size());
        while (itr2.hasPrevious()) {
            System.out.println(itr2.previous());
        }
        // The listIterator is still alive--
        System.out.println(itr2.next());
    }
}
The two most important classes that implement the List interface are **LinkedList** and **ArrayList**.

They have different performance for large lists.

Both have extra methods over and above the List interface.

But by far their most important methods are in the Collection plus List interfaces.

Because of this, it is common to use List type instead of ArrayList:

```java
List<BankAccount> myList = new ArrayList<BankAccount>();
```
Here is a 4 member list:

We can get(0), ..., get(3) and access any particular object ref.
We can set(0, b) and replace the object at 0 with b.
What happens if we set(4, b)?
To grow the list we need to use add(Object x), but where does it go??
This is fast because the LinkedList tracks the end-of-list.
A ListIterator starting from 0, has a next method that returns element 0 on first call, element 1 on second, etc.

Should test with hasNext before doing a next.

If hasNext returns false, the iterator is at end of list (EOL).

It starts at beginning of list, so there are 5 different iterator states for 4 elements:

\[
< A_0 > < A_1 > < A_2 > < A_3 >
\]

\[\uparrow\] original iterator, before element A0

\[\uparrow\] after first next, returning A0 (just before element 1)

\[\uparrow\] after 2nd next, returning A1

\[\uparrow\] after 3rd next, returning A2

\[\uparrow\] after last element
You can think of an iterator as sitting *between* two elements, so to speak.

At each point in time, an iterator is positioned just after the element that would be returned by `previous()`, and just before the element that would be returned by `next()`.

A `ListIterator` can go both ways.

When we talk about numerical position in a list, it’s normally about the position of an element, not directly the iterator.
With the listIterator(int pos) method, the pos determines an element position and the method returns an iterator positioned prior to that element, or at EOL if pos == size of list.

listIterator(0) gives an iterator positioned before the very first element, etc.

A special case is that extra position after all the elements: this is attained by using N as an arg, the number of elements in the list.

**Remember:** an iterator has N + 1 possible positions for N elements.
What happens if next returns A1, then another next returns A2, and then a previous is done – is A1 returned?
What happens if next returns A1, then another next returns A2, and then a previous is done – is A1 returned?

No! We’ve just gone past A2 one way, and now we go back across it again, so A2 gets returned again.
class TestArrayList4
{
    public static void main( String [ ] args )
    {
        ArrayList<Integer> lst = new ArrayList<Integer>( ) ;
        lst.add( 2 ) ;
        lst.add( 4 ) ;
        ListIterator<Integer> itr1 = lst.listIterator( 0 ) ;
        System.out.print( "Forward: " ) ;
        while( itr1.hasNext() )
            System.out.print( itr1.next( ) + " " ) ;
        System.out.println( ) ;
        System.out.print( "Backward: " ) ;
        while( itr1.hasPrevious() )
            System.out.print( itr1.previous( ) + " " ) ;
        System.out.println( ) ;
        ListIterator<Integer> itr2 = lst.listIterator( lst.size( ) ) ;
        while( itr2.hasPrevious() )
            System.out.print( itr2.previous( ) + " " ) ;
        System.out.println( ) ;
        System.out.print( "Forward: " ) ;
        for( Integer x : lst )
            System.out.print( x + " " ) ;
        System.out.println( ) ;
    }
}

This and other List examples are available in TestArrayList.zip
Here the ListIterator<Integer> goes all the way down the list to EOL, then back along the list, so the turn-around occurs at EOL.

Another ListIterator<Integer> starts from “lst.size()”, which would be 4 for our list.

This is an artificial element number denoting the EOL position of the iterator.

Again, there are n+1 different iterator states for n elements, and these are numbered from 0 to n.
• An iterator sits between elements.
• When calling remove, which nearby element gets removed?
• The object removed is the last one returned by next or previous, and only one remove per movement-action is allowed.
  ✓ This also holds for iterators over Sets (in fact, any Iterator)
• What happens if you next, remove, and then next again?
• You access the element just after the removed element. Because we’ve moved past the deleted element already, the iterator position is clear.
• If you next, remove, previous, you should get the previous-to-removed. And so on, using the model above.
• With an iterator going across a Set (entrySet for a Map, end of last class), we saw that a separate Set remove invalidates the iterator, causing an ConcurrentModificationException when the iterator is used again.
• Note that this means the iterator (one object) and the set itself (another object) are in communication (via shared memory)
• Removes made “through the iterator”, i.e. by Iterator’s remove, are not so disruptive: the iterator is still valid.
• There are similar behaviors with iterators on Lists
• But one iterator doesn’t notify another iterator when it does a remove, so we have to be careful with multiple iterators on a List…
Remove List Duplicates With Two Iterators?

- Example: Using two iterators to remove duplicates from a LinkedList doesn’t work!

- Proposed Algorithm:
  - Scan LinkedList list with iterator itr1
  - For each itr1-position, with element o1, initialize a ListIterator (itr2) at that position.
  - Scan rest of list with itr2, removing elements that equal o1.

- The exception is ConcurrentModificationException, when we go to step itr1 to a new position for the outer loop.

- That itr2 mod was communicated to the list object but its details are kept in the itr2 object, not available to the itr1 object
List<Order> list = new LinkedList<Order>(); // or ArrayList
// add some elements to the list: 100, 200, 100, 400
Iterator<Order> itr = list.iterator();
System.out.println("about to do next() in outer loop... list is" + list);
Order o1 = itr.next();
System.out.println("outer loop o1 iterator working on " + o1);
ListIterator<Order> listItr = list.listIterator(position + 1);
while (listItr.hasNext()) {
    Order o2 = listItr.next();
    System.out.println("inner loop o2 =" + o2);
    if (o1.equals(o2)) {
        System.out.println("removing o2 =" + o2);
        listItr.remove(); // this works OK
    }
}
System.out.println("Now advance outer loop o1 iterator...");
if (itr.hasNext())
    itr.next(); // this throws!!
Output:
about to do next() in outer loop... list is[100, 200, 100, 400]
outer loop o1 iterator working on 100
inner loop o2 =200
inner loop o2 =100
removing o2 =100
inner loop o2 =400
Now try to advance outer loop o1 iterator...
Exception in thread “main” java.util.ConcurrentModificationException at
java.base/java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:970)
at java.base/java.util.LinkedList$ListItr.next(LinkedList.java:892) at
ListRemove.main(ListRemove.java:34)
Remove Duplicates With Two Iterators: Notes

- Note that for this example, the Order object needs an equals method.
- Otherwise, the code would not find duplicates at all!
- We can just test the order id between two Orders.
- We know the list objects are Orders because this is a List<Order>, so have int getId() by the Order API.
- See the code on p. 103 for an example of equals.
- When you code equals, you should also code hashCode.
- Use @Override to make compiler check method really overrides.
public class Order {
    private int id; // unique identifier, basis of equality
    // other fields
    public Order(int i) { id = i; }
    public int getId() { return id; }
    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (this.getClass() != other.getClass())
            return false;
        Order o = (Order) other;
        return id == o.id;
    }
    @Override
    public int hashCode() { return Integer.valueOf(id).hashCode(); }
    public String toString() { return "" + id; }
}
Safely Removing Duplicates

- Drop the outer iterator, just get(i) the value of the ith element into a variable.
- Run the inner loop with an ListIterator over the rest of the list, removing all elements that are equal to that value.
- What to do with huge lists, when using get and/or remove in inner loop means $O(n^2)$?
Safely Removing Duplicates

- Drop the outer iterator, just get the value of the element into a variable.
- Run an internal loop with an iterator, removing all elements that are equal to that value.
- What to do with huge lists, when using get and/or remove in inner loop means \( O(n^2) \) or worse?
- Abandon lists!
- You can use HashSet \( h = \text{new HashSet(list)}; // \text{Set means no dups, } O(n) \) (good trick for quick deduplication)
- Then put result back in a list, also \( O(n) \).
- Another way: toArray, then sort, then pick off unique values, \( O(n \log n) \)
Performance of LinkedList vs. ArrayList

For ArrayList of size $n$
- Get, set are very fast, $O(1)$
- Append-type add is fast most of the time. If it involves array expansion, it is expensive, $O(n)$.
- Delete is expensive unless it is at the end.

For LinkedList of size $n$
- Get, set depends on the index position
  - $\text{get}(1)$ is done by two next's down the list from the beginning of the list, and $\text{get}(n-2)$ is done by two previous's from the end of the list
  - Most expensive is $\text{get}(n/2)$
- Remove is easy once the right spot in the list is located. Remove in an iterator is $O(1)$, but the larger task may involve up to $O(n)$ nexts/previous's to get the iterator positioned initially. List.remove(Object o) removes the first matching element.
• Queues and Stacks are specialized lists: they keep elements in a kind of order, and tolerate duplicates.
• Bags are similar, but do not keep order, but do tolerate duplicates
• Bags, Queues, and Stacks are covered in S&W Section 1.3, hopefully covered in cs210
• There they are implemented directly in Java, not using the JDK.
• But the APIs (listed on page 121) are as expected for classic Queues, Stacks, and Bags.
A Stack is a specialized List where we insert (push), retrieve (top or peek), and delete (pop) elements at one end.

A Queue is a specialized list where we insert at one end (enqueue), retrieve and delete (dequeue) at the other.
There is a Stack\(<E>\) class in the JDK, but its performance is not good.
The JDK docs suggest use of ArrayDeque (interface Deque) instead
A Deque\(<E>\) ISA Collection, so this is a "wide" interface (lots of methods)
We can create a Stack\(<E>\) class and implement it with a LinkedList\(<E>\) or ArrayList\(<E>\) to stick to its textbook model.
S&W have a Stack\(<\text{Item}>\) class (API on page 121, code on page 149) based on an singly-linked list
Queue Support in the JDK

- There is a Queue\(<E>\) interface in the JDK.
- The JDK docs list various implementing concrete classes including ArrayDeque.
- Queue\(<E>\) ISA Collection\(<E>\), so this is a "wide" interface.
- In fact, a Queue ISA Deque ISA Collection.
- So a Queue is implemented easily with an ArrayDeque.
- We can create a Queue class and implement it with a LinkedList or ArrayList to stick to its textbook model.
• We can see that a LinkedList or ArrayList often is working inside another object, providing the needed apparatus
• Why use a Queue instead of the List inside it?
• Because it’s easier to reason about code that’s working with a queue than with a List
• We have seen that a List is pretty complicated, with two-way iterators and different ways to remove.
• It’s a little like driving a shift car vs. automatic or self-driving
• In general, we like to use the simplest data structure that has the capabilities we need.
• Note homework 2 problem on using a LinkedList to implement Bag.
Using LinkedList/ArrayList...

• Can we implement a Set<E> with LinkedList<E>?
  
  • Yes, but it won’t perform like a HashSet or a TreeSet, because of the work of avoiding duplicates, and the cost of traversing the list to determine contains().

• Can we implement a Map<X, Y> with a LinkedList<E>?
  
  • Yes, use E to hold a Map.Entry<X, Y>. A get will involve a scan of the list, very inefficient.

• So in practice we use LinkedList and ArrayList to implement “list-like” data structures, where keeping elements in a certain order can be important, and allowing duplicates.

• FYI: The Lisp language uses lists as the primary data structure.
Often data are processed in a specific order, the priority

Examples

- In OS process scheduling, a process comes with a priority, and high priority processes are executed earlier than low priority ones.
- In network routing, high priority traffic (VoIP, IPTV) are delivered before other traffic.

We need to dynamically maintain the pending jobs

- New jobs keep coming in.
- When a server becomes available, the highest-priority job is removed from PQ and serviced.

We’ll defer studying Priority Queues until later in the term.