Linear Containers

- Introduction to abstract data types (ADTs)
- LinkedList class
  - comparison with arrays / ArrayList
  - interface
  - iterators
- Stack ADT
  - applications
  - interface for Java Stack
  - ex use: reverse a sequence of values
  - representations
- Queue ADT: Section 15.5.2
  - Also related Self-check exercises and Interactive Review and Practice exercises

Announcements

Abstract Data Types (ADTs)

- An abstract idea of a data structure
- Can be implemented with a class
- ADT operations = class methods
- Some ADT examples:
  List, Stack, Queue, Set, Map
- Some concrete data structure examples:
  partially-filled array, linked list, hash table.
- The ADT is implemented as a class that encapsulates a concrete data structure, and often has more than one possible implementation
- Can also be modeled using Java `interface` feature and classes that implement the interface (will discuss next lecture)

Today: linear containers

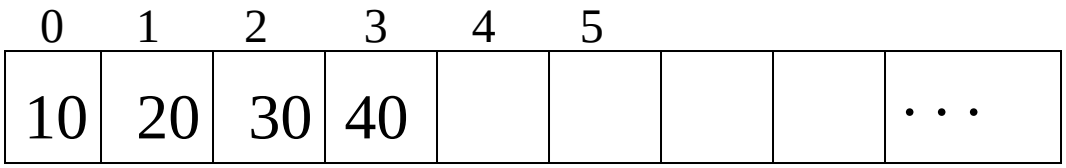
- linear containers: ones where the elements are in a certain sequence: the client can control where the elements go in that sequence.
- Later we'll talk about other ADTs where the collection is not in a specific sequence and the ordering is not controlled by the client (e.g, Set).

List ADT

- **`List`** is a Java *interface* with two implementations:
  - **`ArrayList`**
  - **`LinkedList`**
- because of different performance characteristics they aren't very interchangeable.

Review: array / ArrayList

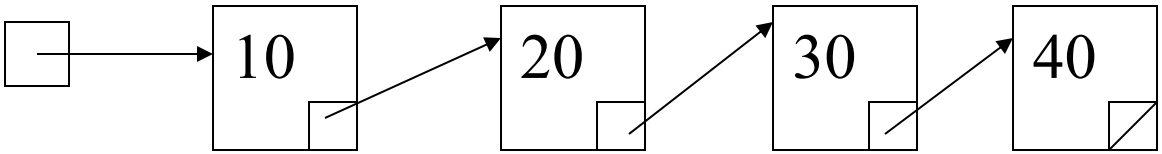
- Want to store a collection of things (elements) in some order.
- All elements are the same type
- Want random access to elements
- Can use an array:



- If number of elements to store is unknown: partially-filled array (or ArrayList)

Introduction

- Alternate: linked list
  - Only use as much space as you need at a time.
  - Can insert and delete from middle, as well as front or end, without shifting values left or right by one.
  - However *no* random access based on location. E.g., get element at position **`k`** is not constant time:
    - has to traverse to element **`k`**



Linked list implementations

- Will writing our *own* linked list code later this semester (using C++)
- Java (and C++) has a LinkedList class:

  `LinkedList<ElementType>`
- has some of the same methods as **`ArrayList`**
- but, WARNING, some of them run slower. E.g.,

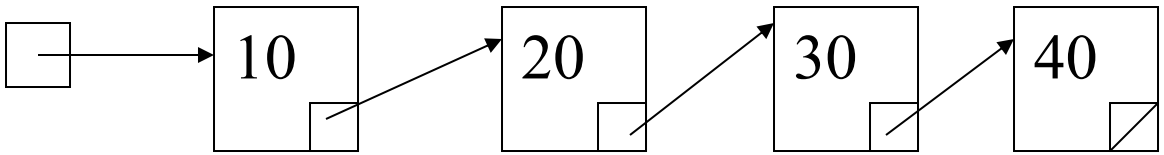
```
list.get(i)
list.set(i, newVal)
```

Using ArrayList methods with LinkedLists

(these methods are legal on LL: part of List interface)

```
void printList(LinkedList<Integer> list) {
    for (int i = 0; i < list.size(); i++) {
      System.out.println(list.get(i));
    }
  }
```

- What is the big-O time to run this code?



Asynchronous participation: Link to **`printList`** poll

Using ArrayList methods with LinkedLists (cont.)

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

- A bad way to traverse a linked list.

- Generally, in a loop to traverse, avoid using the methods that take an index:
  e.g., get(i), add(i, object), remove(i), set(i, object)

Putting elements in a LinkedList

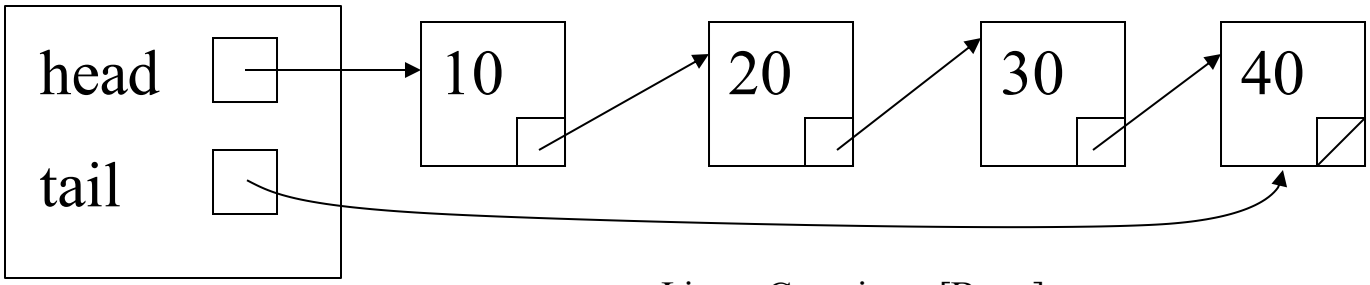
- Create an empty list:

```
LinkedList<Integer> list = new LinkedList<Integer>();
```

- Put some stuff in the list:

```
list.add(10);
list.add(20);
list.add(30);
list.add(40);
```

- Adding to the end (or beginning) is efficient: O(1)
- Internally uses a "tail" pointer (or equivalent)



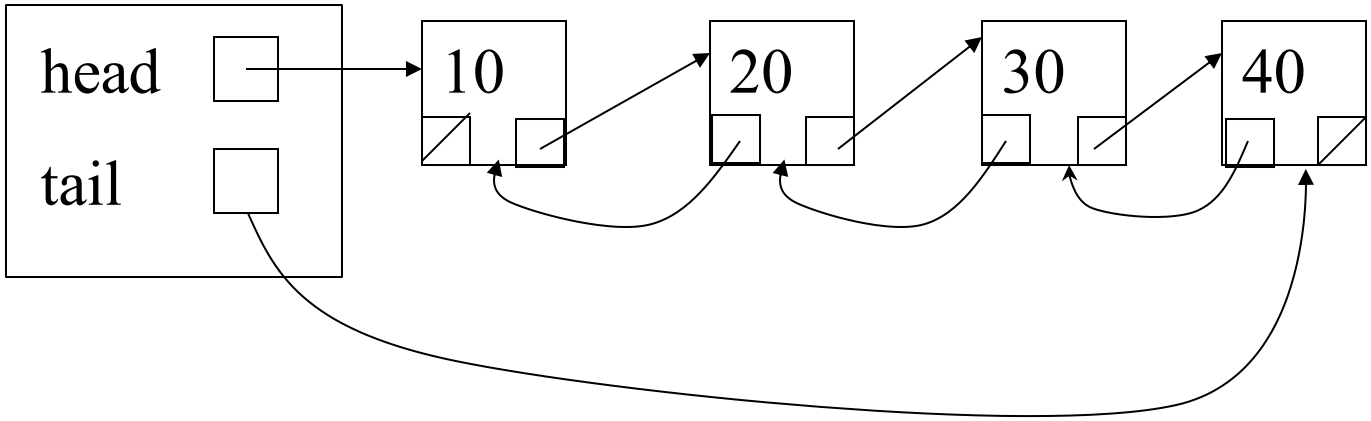
Some LinkedList-specific methods

- Operations on the beginning or end are constant time:

```
// suppose list initially contains :
        [Aarti, Sally, Min, Bao]

list.addFirst("Gaga");

list.getFirst()   // returns Gaga

list.getLast()    // returns Bao

list.removeFirst();   // removes Gaga

list.removeLast();    // removes Bao
```

Doubly LinkedList with Head and Tail pointer

- All of the "end" operations are efficient because list actually has the structure:

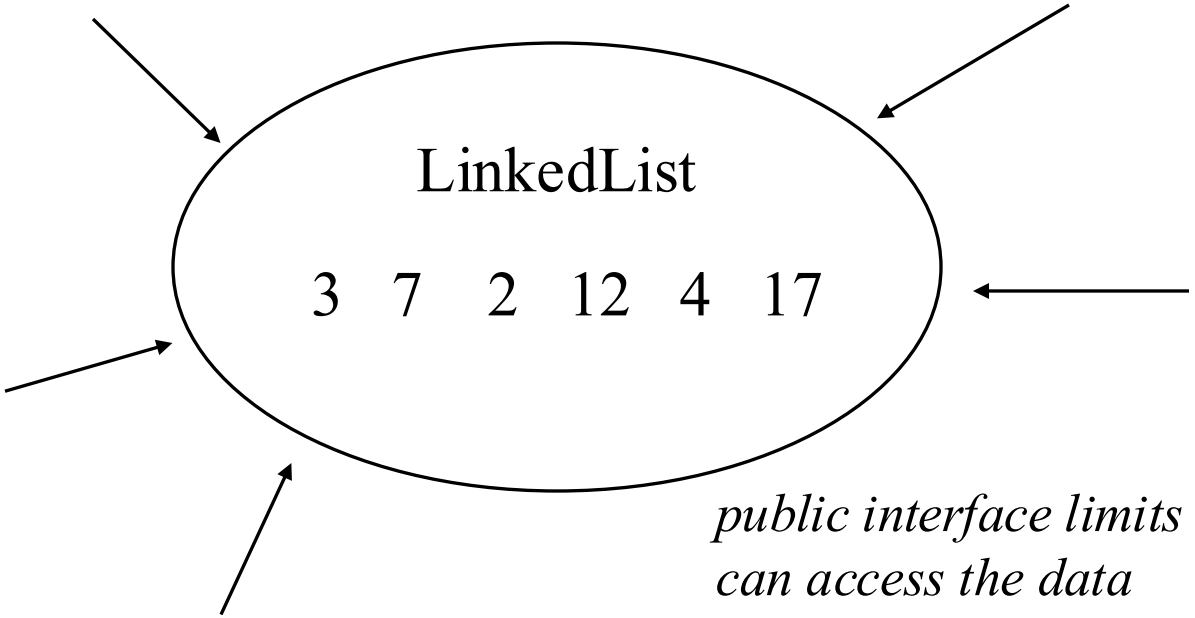


So, how *do* we traverse a LinkedList?

- Recall: **`for`** loop with **`get(i)`** is a **`bad`** idea.
- To traverse, use a **`ListIterator`** object
- Associate it with a particular list
- Abstracts the idea of some position in the list
- We can also use it to add or remove from the middle.
- Iterators apply to more than just LLs:
  - can also use ListIterator to traverse an ArrayList
  - will also use iterators for other container classes

Why we need iterators

- Container object encapsulates a bunch of values
- But sometimes we need a way to get to each of the individual values hidden inside.



public interface limits how client can access the data

ListIterator

- Iterator interface is similar to **`Scanner`**:

  `next()`

  `hasNext()`

- Guard calls to **`next()`** with a call to **`hasNext()`** so you don't go past the end of the list
- [If you do **`next()`** when **`hasNext()`** is **`false`**, crashes with **`NoSuchElementException`**]
- To get an iterator positioned at the start of **`list`**:

```
ListIterator<String> iter = list.listIterator();
```

ListIterator

- Iterator points between two elements.
- 5 possible positions for iterator on the following list:

```
[Aarti, Sally, Min, Bao]
```

Traversing with a `ListIterator`

```
// print out all the elements of the list:
ListIterator<String> iter = list.listIterator();
while (iter.hasNext()) {
    String word = iter.next();
    System.out.println(word);
}
```

`next()`: returns the element after iter position and advances iter beyond that element

Suppose **`list`** contains:

`[Aarti, Sally, Min, Bao]`

`next()` changes state of iterator

- Want to print out all values >=60
- Suppose list contains:

```
[33, 94, 56, 59, 65]
```

- What is the output of the following code:

```
ListIterator<Integer> iter = list.listIterator();
while (iter.hasNext()) {
   if (iter.next() >= 60) {
      System.out.println(iter.next());
   }
}
```

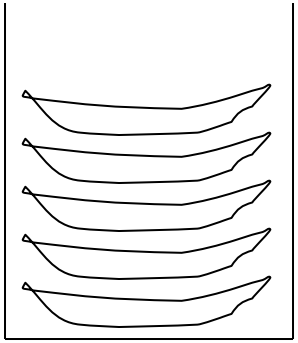
Asynchronous participation: Link to **`ListIterator`** poll

Other `ListIterator` methods

- you can do a reverse iteration with ListIterator
- you can add or remove elements from the list with ListIterator.
  - E.g., traverse to the element you want to remove, then use iterator to remove that element.
- See textbook for details (Section 15.2.3)

Stacks

- a collection of things (like an array is)
- but with restricted access
- the only item you can look at or remove is the *last* item you inserted. Last In First Out (LIFO).
- E.g. stack of dishes
  - push a plate on the top of the stack
  - pop a plate from the top of the stack
  - examine the plate at the top of the stack
  - ask if the stack is empty



Stacks for method call/return

- one example of a stack is the *system stack* (a.k.a., run-time stack, or call stack)
- one element is called a *stack frame* (aka, *activation record*):
  - all the *data* associated with that call: e.g., locals, params, return addr.
- method call/return follows LIFO order:
  - calling a method: push "method" onto stack
  - returning from a method: pop "method" from the stack
- last method called will return before any methods that called it.

Other stack applications

- Alternative to using recursion in some recursive algorithms: e.g, DFS
- CS Theory of Computation: some categories of machines
  - Finite Automata (DFA) – state machine (recognize regular langs.)
  - Pushdown Automata (PDA) – state machine plus one stack (recognize CFLs)
  - Turing machine – equivalent to general purpose computer
- (Related) Compilers / Programming languages:
  - syntax described with CFG

    <expr> → *<expr> <addSubOp> <term>* | *<term>*
    <term> → *<term> <mulDivOp> <factor>* | *<factor>*
    <factor> → *- <factor>* | *<var>* | *<literal>* | *(<expr>)*
  - parser is a PDA (parser generator converts the CFG to a PDA to recognize that language)
  - Some parsers use human-coded recursion instead

Java **`Stack`** class

```
import java.util.Stack;

Stack<Integer> s = new Stack<Integer>();
                        // creates an empty stack

s.push(3);              // add an element to the top of
                        // the stack

int n = s.peek();       // returns top element in the
                        // stack (does not modify stack)

int top = s.pop();      // pops top element off of
                        // stack and returns it

s.empty();              // tells whether stack is empty
```

Using **`Stack`** operations

```
Stack<Character> s = new Stack<Character>();
         // creates an empty stack of characters

s.push('a');
s.push('b');
s.push('c');
System.out.println(s.peek());

s.pop();
s.push('d');
System.out.println(s.peek());

s.pop();
s.pop();
System.out.println(s.empty());
```

Ex: Reversing a sequence

- Stacks are good for reversing things
  - The last item you put on is the first one you get out
  - The second to last item you put on will be the second item you get out, etc.
- Example problem: read in a bunch of integer values (stop when **`hasNext()`** is false) and print them out in reverse order.

```
void reverse(Scanner in) {
```

Efficient Stack representations

- If you were writing a Stack class yourself . . .
  - How to represent?
  - Where should top be?
  - What is big-O of each operation?