Linear Containers

- Introduction to abstract data types (ADTs)
- LinkedList class
  - comparison with arrays / ArrayList
  - iterators
- Stack ADT
  - applications
  - interface for Java Stack
  - ex use: reverse a sequence of values
  - representations
- Queue ADT
  - applications
  - interface for Java Queue
  - representations
- Time permitting: additional example of implementing an interface

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:
  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

List ADT

- No standard definition, but it is an **`interface`** in Java.
- Java **`ArrayList`** and **`LinkedList`** implement the List interface.
- However, because of different performance characteristics between the two of them, they aren't really interchangeable.

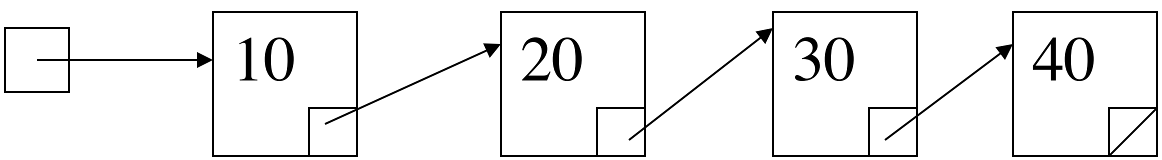
Review

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

0	1	2	3	4	5			
10	20	30	40					. . .

Introduction

- Alternate: linked list
  - Only use as much space as you need at a time.
  - Can insert and delete from middle 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 discuss code for writing our *own* linked list code later this semester (using C++)
- Java (and C++) has a LinkedList class:

```
LinkedList<ElementType>
```

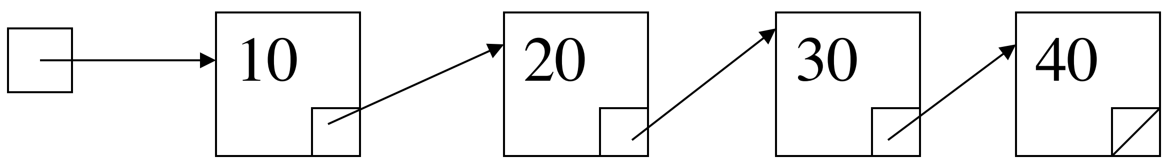
- has some of the same methods as **`ArrayList`**
- but, WARNING, some of them run slower. E.g.,

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

Using ArrayList methods with LinkedLists

```
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?



Using ArrayList methods with LinkedLists

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

- A bad way to traverse a linked list.

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

Putting elements in a LinkedList

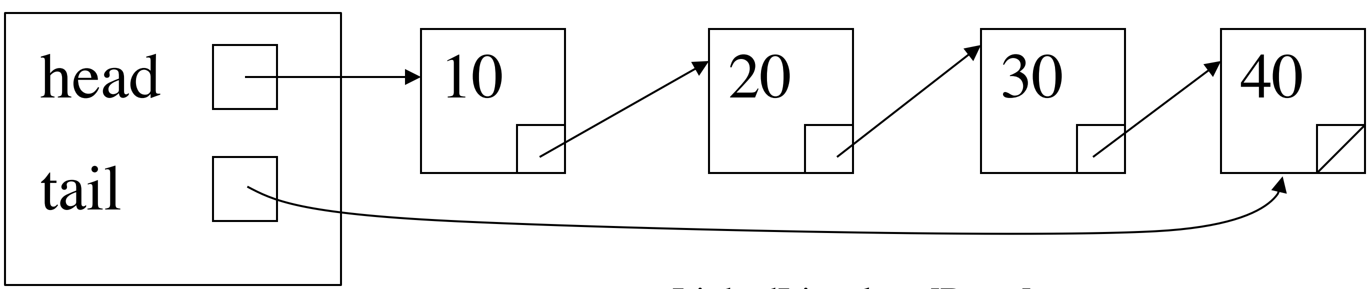
- Create an empty list:

```
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)



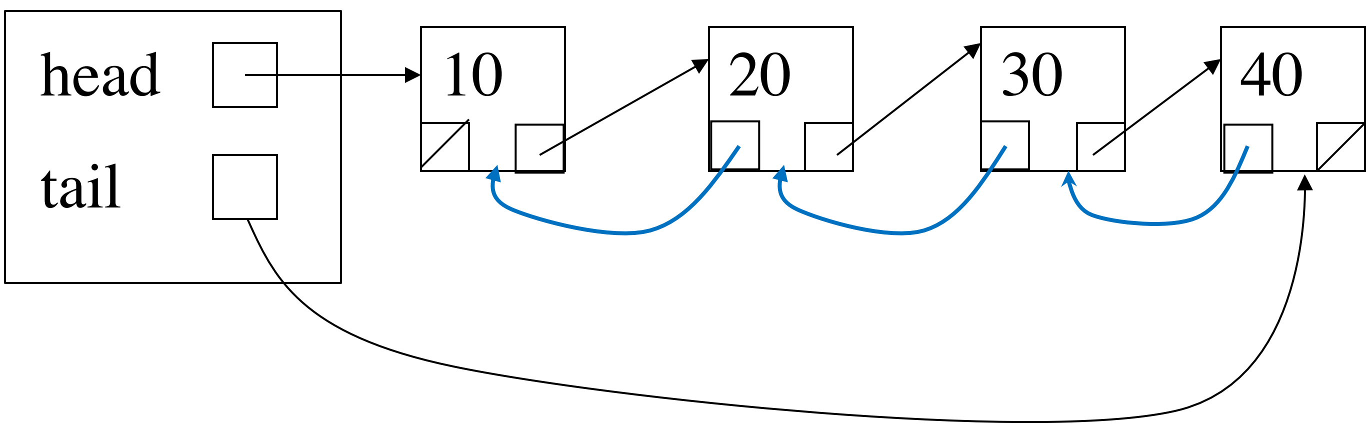
Other LinkedList methods

- Operations that access the beginning or end are efficient:

```
// suppose list contains :
        [Anne, Sally, George, Carol]

list.addFirst("Gaga");

list.getFirst()   // returns Gaga

list.getLast()    // returns Carol

list.removeFirst();   // removes Gaga

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

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 are a more general idea:
  - can also use ListIterator to traverse an ArrayList
  - will also use iterators for other container classes

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
- 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:

```
[Anne, Sally, George, Carol]
```

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);
}
```

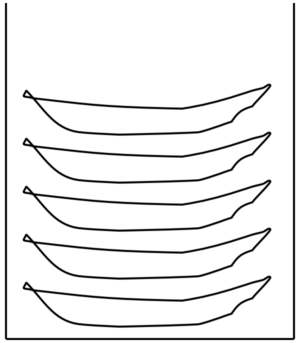
`next():` returns the element after iter position and advances iter beyond that element

Suppose **`list`** contains:

`[Anne, Sally, George, Carol]`

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 ones that called it.

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 and then print them out in reverse order.

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

Stack representations

- How to represent?
- Where should top be?
- What is big-O of each operation?

Queue

- A container of elements that can be accessed/inserted/removed like standing in line.
- Enter queue at the end (enqueue)
- Exit queue at the front (dequeue)
- Can access front element
- First-in First-out (FIFO)

Applications of Queues

- Major application is to represent lines (queues) in software.
- E.g., big in operating systems
  - print queue – print jobs waiting to print. They are processed in FIFO order.
  - process queue – processes waiting for their turn to execute.

Queue applications (cont.)

- also in simulations
  - queue of events waiting to be processed
  - simulating queues from the world we are simulating (e.g., Bank line, airplanes waiting to take off)
- and Java GUI system
  - queue of user input events waiting to be processed
  - (e.g., mouse clicks, keyboard strokes)

Java **`Queue`**

- **`Queue`** is an **`interface`** rather than a class.
- **`LinkedList`** implements this interface.
- To create one:

```
Queue<MyType> q = new LinkedList<MyType>();
```

- means we intend to only use the LL ops specified by **`Queue`**.

Java **`Queue`** interface

```
import java.util.Queue;
import java.util.LinkedList;
Queue<Integer> q = new LinkedList<Integer>();
             // creates an empty queue of integers

q.add(3);           // add an element to the end

int n = q.peek();  // returns first element in
              // the queue (does not modify queue)

int front = q.remove();
                  // removes first element from queue
                  // and returns it

q.isEmpty();        // tells whether queue is empty
```

Using **queue** operations

```
Queue<Character> q = new LinkedList<Character>();

q.add('a');
q.add('b');
q.add('c');
System.out.println(q.peek());

q.remove();
q.add('d');
System.out.println(q.peek());

q.remove();
q.remove();
System.out.println(q.isEmpty());
```

Queue representations

- How to represent?
- Where is front and end?
- What is big-O of each operation?

Additional example of implementing an interface

- Problem: sort an array of **`Rectangle`**'s in increasing order by area.
- Do not implement your own sort method!

```
public static void sortIncrByArea(
                        Rectangle[] rects) {

    Arrays.sort(
```