- full implementation
- circular linked lists
- recursion and linked lists
- recursion exercises

Next week and homework

- next week: intro to binary trees
- homework by next week:
  - download, run and understand this week's example programs
  - read Horstmann, sections 17.1 – 17.2

Lab

- lab assigned in 'Implementation of linked lists'. See Canvas for due date

Review last week

- finally did the simple dynamic implementation of linked lists, using references
  - saw that all the hard work is done during design, drawing pictures
  - Java syntax for references is straightforward
  - Java strict type checking at compile time makes code safe and easy to understand
- looked at double linked lists
  - more powerful than single linked – move both directions; delete node at p
  - more generally used
- as a quick warm-up exercise, let's append a new item to the end of a single linked list e.g.

```
l.append(10);
```

  - draw the picture

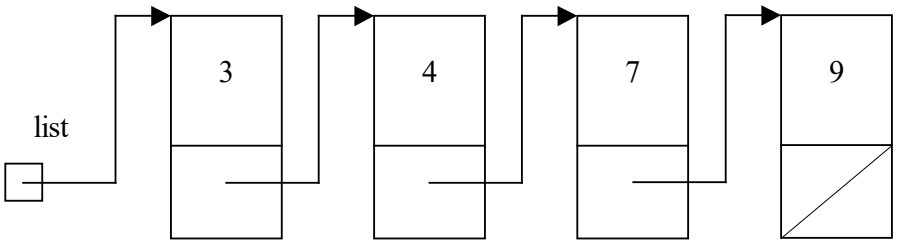

Figure 1 a single linked list

  - find two special cases, of list is empty or !empty
  - can write the algorithm in pseudocode, something like:

```
create the new item r
if list is empty
   list = r
else
   p = list
   while (p.next != null)
      p = p.next
   p.next = r
```

- with more experience with the language, you will find that your pseudocode becomes Java, so you can write the method directly:

```
public void append(int i)
{
    Item r = new Item(i);
    if (isEmpty())
        list = r;
    else {
        p = list;
        while (p.next != null)
            p = p.next;
        p.next = r;
    }
}
```

Introduction to this week

- finish linked lists. Extend last week's simple implementation to our usual, more general full implementation
  - syntax is straightforward
  - but see some conceptual problems
- for practice, look at a refinement of a simple linked list
  - allows easy access to front and rear of a list
- begin to introduce next week's highly recursive tree data structure by looking at recursion with linked lists
- some recursion exercises assigned

Full implementation
Objective: build on the simple list implementation from last week, to a full dynamic implementation

Add features to the simple beginning

- as we did for stacks and queues, will now add many features to the simple beginning implementation of list from last week
  - use interface
  - make generic
  - add exceptions
- followed along from when we added these features to the simple stack back in 'Dynamic implementation of stack' module
  - implementation is straightforward, nothing surprising there

Review my `List` example program

- from Canvas, 'Finish linked lists' module, Example programs, download, read, run and understand my `List` example program
  - see in `Tester::main()` that I tested the generic feature by using a list of `Character` for a change
  - then just duplicates the testing of the original simple list implementation
  - then demonstrates some exception handling, by deliberately trying to remove from an empty list
  - works, but some problems…

Not very generic

- there's a conceptual problem here that causes this full implementation to be not very generic
  - the problem occurs because the list interface requires references into the list implementation
  - i.e. in `ListInterface<T>`:

```
public interface ListInterface<T>
{
    . . .
    void insertAfter(Item<T> p, T item);

    T deleteAfter(Item<T> p);

    Item<T> find(T item);
}
```

- see how these methods require references into the list implementation, therefore have to use the `Item<T>` data type

- so this implementation is specific to the implementation of list using a linked list of `Item<T>`, cannot be used if implementing the list in a different way, for example using an array

Use iterator and comparator classes here

- in real life we would use 'iterator objects' to solve this problem

  - an iterator object has a position in a list, so it does the same job as the problematic references above

  - but is a generic class, independent of the underlying implementation data types, so works for any implementation

- similarly, we would also provide generic 'comparator operators' here, that allow us to compare information in the list regardless of its data type, implementation

  - requires two comparisons: `compare()` and `equals()`

  - again, would make this implementation more generic

Summary

- extending our original simple implementation is straightforward

  - but the necessary list concepts of position in a list and comparisons between generic list items would require further refinements

Circular linked lists
Objective: introduce a special case of non-linear linked list

Gives easy access to front and rear of a list

- a disadvantage of a single linked list:

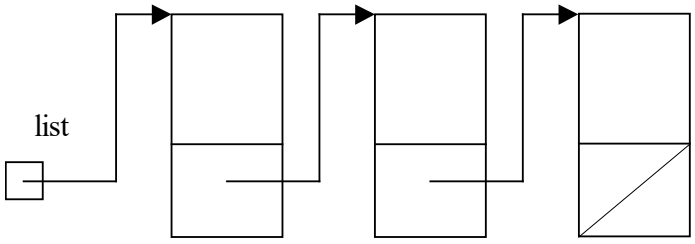

Figure 2 a single linked list

  - it's not efficient to access the rear of a list, if we often need to append new items to the end

- one way to fix this is with a circular linked list:

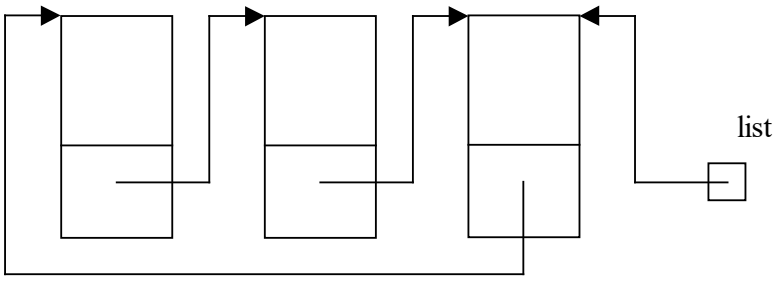

Figure 3 a circular linked list

  - link the rear item back to the front
  - by convention, maintain `list` to reference the rear item, not the front!
  - (BTW, with these changes, will have to rewrite our single linked list methods to work for a circular linked list, soon)

- so with just one reference `list`, now have easy access to:
  - front - is `list.next`
  - rear - is `list`
  - e.g. could implement a queue using a circular linked list!

- will implement some interesting circular list methods, for practice

`append()` method

- adds a new item to the end of a list e.g.

```
CircularList cl = new CircularList();

. . .

cl.append(999);
```

- draw the pictures to see all cases of empty list, and !empty list:

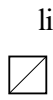

Figure 4 an empty circular linked list

  - see that the definition of empty for a circular linked list remains the same, is simply when list is null

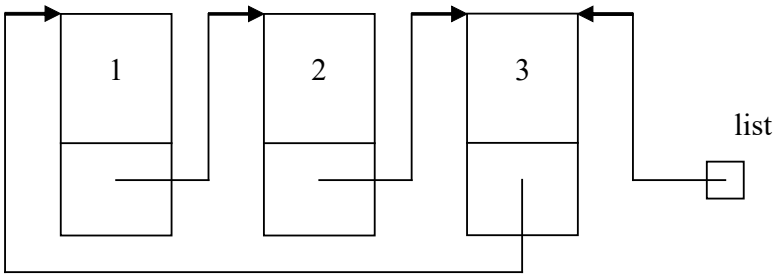

Figure 5 a !empty circular linked list

- from the pictures, algorithm is something like:

create the new item
if list is empty
  link the new item back to itself
else
  link new item to end of existing list
update list to reference new item

- this gives something like:

```
public void append(int x)
{
    Item r = new Item(x);
    if (isEmpty()) {
        r.next = r;
    }
    else {
        r.next = list.next;
        list.next = r;
    }
    list = r;
}
```

`toString()` method

- as usual, must traverse the list to build a comma separated string of items e.g.

```
System.out.println(cl.toString());
```

  – for example, output format would look like this:

```
1, 2, 3, 4, 5
```

- draw the pictures to see all cases of empty list, one item and more than one item in the circular linked list:

Figure 6 an empty circular linked list

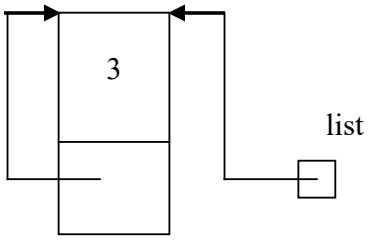

Figure 7 a circular linked list containing one item

  – see here how the single item in a circular linked list has to link back to itself

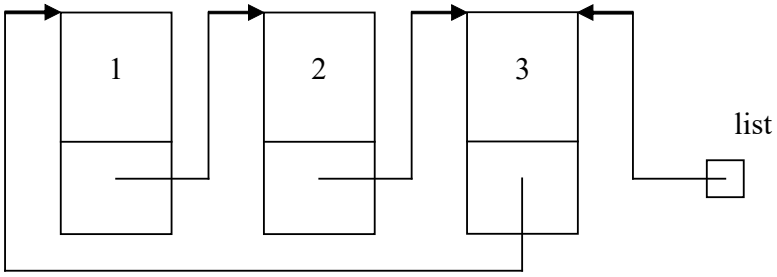

Figure 8 a circular linked list containing more than one item

- have to be careful in all cases for no comma after last item

- from the pictures, algorithm is something like:

  create an empty string
  if list is !empty
    start at beginning of circular list
    while not the last item
      append item followed by ", "
      move to next item
    append last item, no comma
  return string

- this gives something like:

```
public String toString()
{
    StringBuilder s = new StringBuilder("");

    if (!isEmpty()) {
        Item r = list.next;
        while (r != list) {
            s.append(r.info + ", ");
            r = r.next;
        }
        //append last item
        s.append(r.info);
    }
    return s.toString();
}
```

  - use the `StringBuilder` library class for efficiency as usual

Another solution

- BTW, another solution would be to simply add another reference, `rear`, to the list e.g.

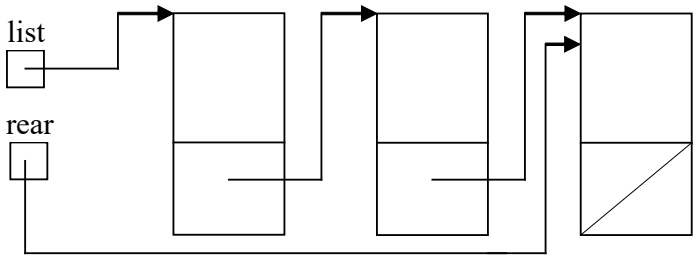

Figure 9 an easier way to have access to front and rear of a list

- simpler and clearer than a circular list

Summary

- from Canvas, 'Finish linked lists' module, Example programs, download, read, run and understand my `Circular list` example program

Recursion and linked lists
Objective: lists are recursive data structures, so can be processed recursively

A linked list can be defined recursively

- a linked list can be defined recursively as "a list reference which is null, or which points to an item, which contains information and a list reference"
  - looks like recursion in the declaration of `next` in an `Item`:

```
public class Item
{
    protected int info;
    protected Item next;

    . . .
```

Methods can be implemented recursively

- so could implement some list processing recursively e.g. a recursive traversal

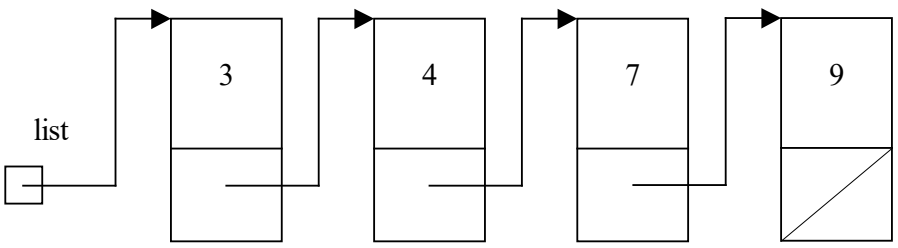

Figure 10 a linked list

  - visit each item in turn, say to print it:

```
public void recursivePrint(Item r)
{
    if (r != null) {
        System.out.print(r.info + " ");
        recursivePrint(r.next);
    }
}
```

  - BTW, see that this is tail-end recursion, where no work is done after the recursive method call, so it's more efficient to implement iteratively

Recursion is sometimes the best solution

- …but sometimes a recursive solution is actually the easiest e.g. print the list in reverse order

- this is difficult iteratively – because it's difficult to find the item before the current item in a single linked list

- is easy recursively:

```
public void reverseRecursivePrint(Item r)
{
    if (r != null) {
        reverseRecursivePrint(r.next);
        System.out.print(r.info + " ");
    }
}
```

Review my `List with recursion` example program

- from Canvas, 'Finish linked lists' module, Example programs, download, read, run and understand my `List with recursion` example program

  - (you will write some other recursive list methods for an exercise, shortly)

Summary

- linked lists can be defined recursively, and implemented recursively, BUT

  - recursion is generally less efficient than iteration

  - any recursion can be re-written using iteration

  - so only write recursive solutions where it is easier than iterative

- i.e. will begin trees next week, a very recursive data structure where algorithms are written recursively

Recursion exercises
Objective: practice writing some recursive methods

- practice writing two recursive methods for single linked lists
  - design and write these methods individually, right now
  - then the process is reviewed

1. Recursively count number of items in a list

- e.g.

```
System.out.println("Count is: " + l.count());
```

- hint: start with this `count()` method in the `List` class. This gives us access to the `list` reference, which is private to the `List` class:

```
public int count()
{
    return recursiveCount(list);
}
```

- draw the pictures to see all cases of empty list, one item in list, and more than one item:

Figure 11 an empty linked list

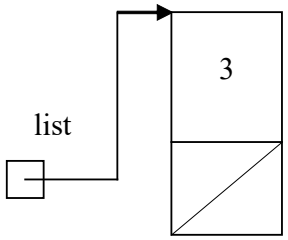

Figure 12 a linked list containing one item

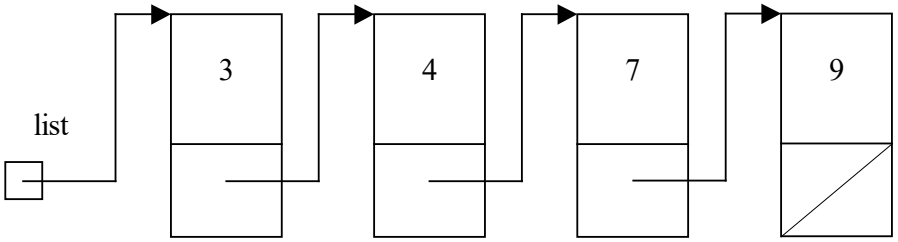

Figure 13 a linked list containing more than one item

- from the pictures, recursive definitions are something like:
  - terminating case: count(list) is 0 if list is empty
  - simpler call: count(list) is 1 + count(list.next)
- this gives:

```
public int recursiveCount(Item r)
{
    if (r == null)
        return 0;
    return 1 + recursiveCount(r.next);
}
```

  - nice

2. Recursively free all items in a list

- (note that this is only an exercise
  - manually traversing a list to free its items is required in languages such as C, C++
  - but memory allocation and deallocation is handled by automatic garbage collection in Java, so something like this is not necessary)
- e.g.

```
l.freeList();
```

- draw the pictures to see the usual cases of empty list, one item in list, and more than one item:

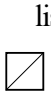

Figure 14 an empty linked list

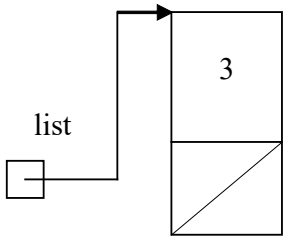

Figure 15 a linked list containing one item

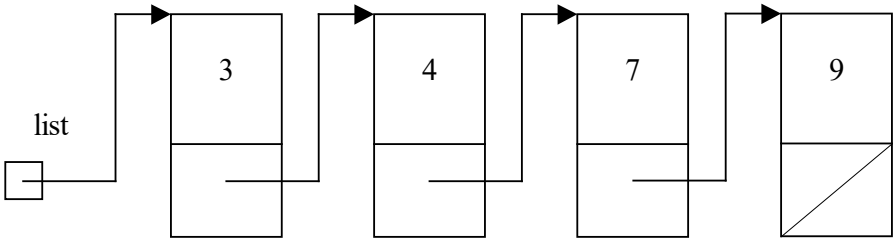

Figure 16 a linked list containing more than one item

- hint: this methods starts with list as null or !null, and must set it to null
- from the pictures, recursive definitions are something like:
  - terminating case: freeList() returns if list is empty
  - simpler call: call freeList() with list = list.next
- this gives:

```
public void freeList()
{
    if (list == null)
        return;
    list = list.next;
    freeList();
}
```

  - BTW, inefficient tail recursion again

Java manages memory automatically

- BTW, if we ever did need to explicitly free a list, it would simply be:

```
public void freeList()
{
    list = null;
}
```

  - then it's Java's responsibility to free all the items in the list
  - conceptually, there is now no reference to the first item in the list, so it can be freed
  - then there is no reference to the next item and it can be freed
  - and so on

- BTW, can use the Java library `Runtime` class to instrument program memory consumption e.g.

```
Runtime rt = Runtime.getRuntime();
System.out.println("rt.totalMemory());
```

  - every Java program automatically gets a single `Runtime` object
  - must first call the `getRuntime()` method, that returns this object
  - then call library methods using this object
  - e.g. `totalMemory()` returns the total amount of memory currently available for current and future objects, measured in bytes
  - as usual, see the Java API docs for full information:

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Runtime.html

- however, the details of memory management are not part of the language specification, is actually controlled by the implementor of the JVM
  - so a programmer can make no assumptions about how the list would actually be freed
  - implementation could vary from platform to platform

Summary

- from Canvas, 'Finish linked lists' module, Example programs, download, read, run and understand my `Recursion exercises` example program

Next week and homework

- next week: intro to binary trees

- homework by next week:

  - download, run and understand this week's example programs

  - read Horstmann, sections 17.1 – 17.2

Lab

- lab assigned in ‘Implementation of linked lists’. See Canvas for due date