- intro to single linked lists
- insert and delete for a list

Next week and homework

- next week: implementation of linked lists
- homework by next week:
  - (no example programs this week)
  - read Horstmann, section 16.1

Lab

- lab assigned in 'Queues'. See Canvas for due date

Review prior work

- covered queues in the week before the Midterm:

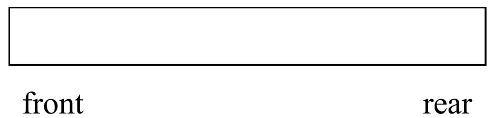

Figure 1 illustration of a queue

  - FIFO data structure
  - remove from the front
  - insert at the rear

- began with a simple single linked list implementation of queue:

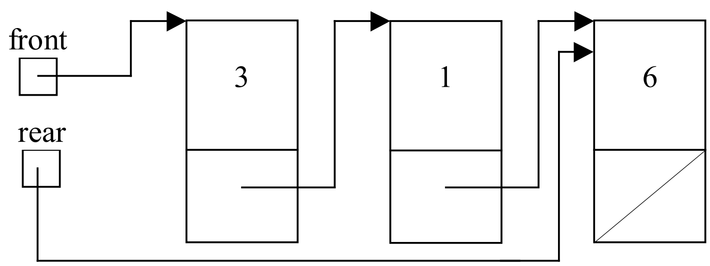

Figure 2 single linked list implementation of queue

  - maintain 2 references `front` and `rear`
  - `front` is next element to be removed
  - `rear` is last element inserted

- queue lab assigned, to wrap queue inside a circular array

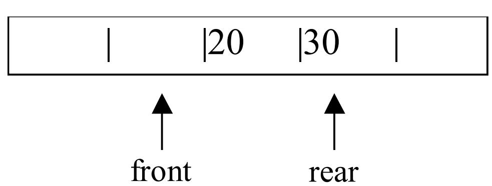

Figure 3 wrap a queue inside a circular array

  - front – location before next item to be removed
  - rear – last element inserted

Introduction to this week

- introduce next new data structure – single linked lists
- will concentrate on the important design technique of drawing pictures to design algorithms
- will work only in pseudocode this week

Intro to single linked lists
Objective: review lists, see some primitive operations

<u>A list is an "ordered sequence of items"</u>
- as we all know, "a list is an ordered sequence of items"
  - e.g. a shopping list:

    bread
    milk
    cereal
    cheese
    butter

  - is ordered: so first, last, next and previous items all make sense
  - unlike a queue, <u>no restrictions on where we insert and delete items</u>
  - …so a list is more general / less restrictive than a queue
  - (note that there is no requirement that items be in any sorted order)

<u>Compare static and dynamic implementations</u>
- we can do <u>static</u> and <u>dynamic</u> implementations of lists
  - static – "size and shape cannot change at runtime" e.g. use an array to implement a list
  - dynamic – "size and shape of data structure can change at runtime" e.g. use references to implement a single linked list
  - different advantages and disadvantages, will review below

<u>Too much memory, otherwise not enough</u>
- with a static implementation, we have to allocate too much memory, otherwise we may not have enough
  - as the size of the list changes, we must always have enough array locations in which to store it
  - therefore, have to allocate in advance the maximum memory required, even though not all of it is used all the time

- a dynamic implementation is more efficient here, as the amount of memory allocated increases and decreases as the size of the list changes

<u>The update problem</u>

- another disadvantage of static implementations occurs when we insert or delete items. Is called the 'update problem' e.g.

  – given a sorted list of numbers implemented statically:

array

2	4	6	8	…

Figure 4 a sorted list of numbers implemented statically, as an array

  – to insert 3 – have to move elements from one memory location to another

  – to delete 2– have to move elements from one memory location to another

  – imagine having to move thousands of large objects!!!!

  – CONCLUDE: insert / delete in a static data structure is very expensive

- a dynamic implementation is very efficient when we insert or delete items

  – here's a picture of the list implemented dynamically as a single linked list of items

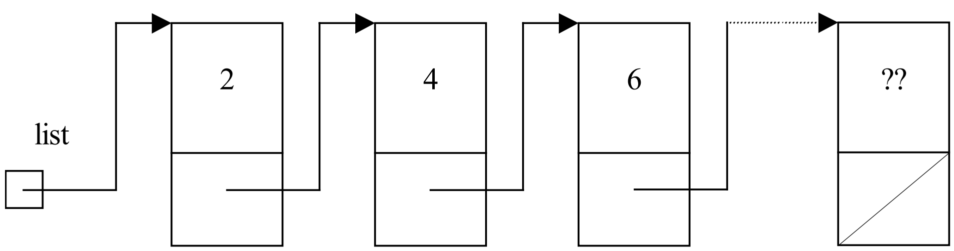

Figure 5 the list implemented dynamically, as a single linked list of items

  – (see how the programmer must maintain a reference to the first item in the list, is named 'list' here)

- do insert and delete in this picture:

  – to insert 3 – only have to update two references

– to delete 2 – only have to update one reference

– never have to move elements from one memory location to another!!!!

– CONCLUDE: insert / delete in a dynamic data structure is much more efficient

Review working with references

- first, remember that our `Item` class has two instance variables:

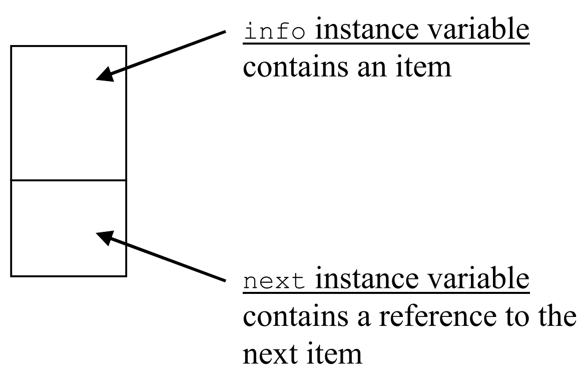

Figure 6 the `Item` class

– `info` is just an `int`, for simplicity

– `next` is the reference to the next item in the list

- we declared these as protected for our convenience, so they can be directly accessed by classes that use the `Item` class

  – (otherwise we would write get and set methods to access instance variables declared private)

- here are pseudocode primitives for working with items, so that we can start to design list operators

- r.info – is the info instance variable of item referenced by r, e.g.

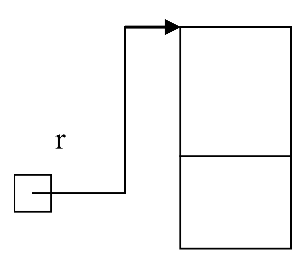

Figure 7 the `item` object referenced by r

- try the effects of the following, updating the pictures (don't worry about syntax here, this is just informal pseudocode):

  r.info = 5
  x = r.info

- r.next – is the next instance variable of item referenced by r

  - e.g. given the following situation, with additional references p, q, r into a list of three items:

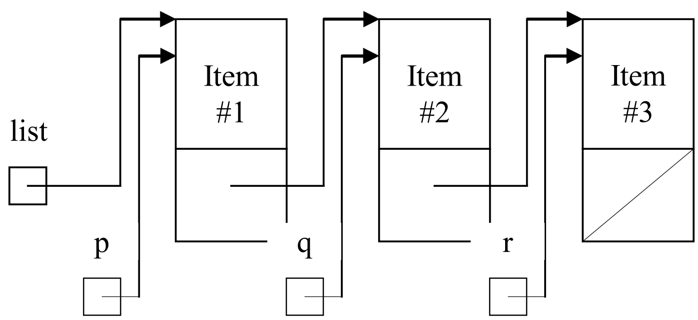

Figure 8 working with references example

  - what do each of the following do? (Put the list back to this starting state each time.) The effect is given after each operation:

  p = q.next - sets p to reference Item #3

  p.next = r - sets Item #1 to reference Item #3. So it deletes Item #2

  p.next = q.next - again sets Item #1 to reference Item #3, removing Item #2

  p.next = r.next - deletes Items #2 and #3 from list

- r = getItem() – creates an item in memory and returns its address for assignment to r

  - e.g.

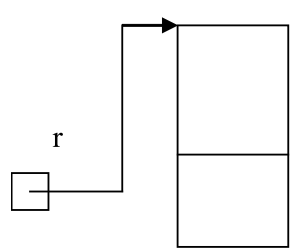

Figure 9 the getItem() primative creates a new item

- r.freeItem() – frees item referenced by r i.e. returns to OS the memory allocated for item referenced by r
  - failing to free memory we previously requested is referred to as a <u>memory leak</u>
  - is bad programming!
  - can cause a program to crash!
  - (note that as Java programmers, we do not have to manually free memory that we have allocated. Instead this is the language's responsibility, implemented as automatic garbage collection)

<u>How to design algorithms</u>

- will do four examples. Remember how to design algorithms for dynamic data structures:
  - draw picture of a start state
  - identify all possible cases
  - use pictures to design algorithms that get to required end state
- (BTW, be sure the algorithms work for empty lists also:)

list

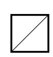

Figure 10 an empty list

1. <u>Insert a new item at the beginning of a list</u>

- two different cases here, for an empty list and a !empty list
  - draw pictures of a start state
  - use this to identify the different cases
  - then use the pictures to figure out steps to get to the desired end state
  - e.g. add item containing 1 to beginning of this list:

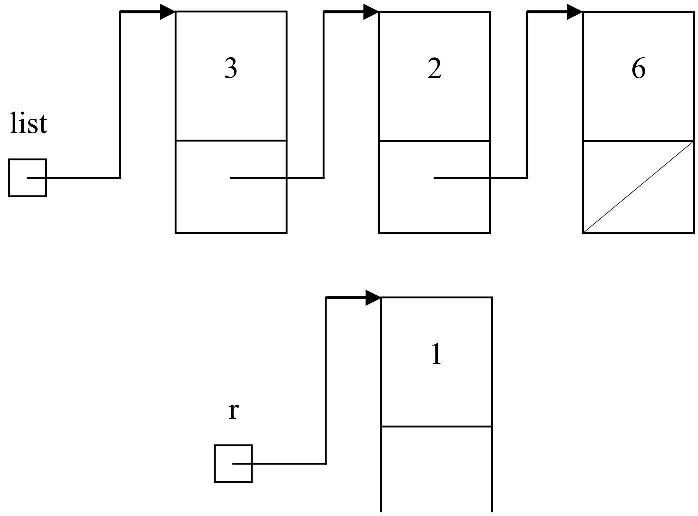

Figure 11 an example of a non-empty list

– or to the beginning of the empty list:

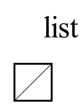

Figure 12 an example of an empty list

- using the pictures, we come up with an algorithm something like this, written as pseudocode. (Update the pictures as you work through the steps):

```
r = getItem()    - create the new item
r.info = 1       - set its info field
r.next = list    - link it into the beginning of the list…
list = r         - …and finished
```

  – see that this algorithm works correctly for an empty list also

2. <u>Remove the first item from the beginning of a list and retrieve its information</u>

- two different cases, for an empty list and a !empty list. Will only consider here the !empty list, e.g.

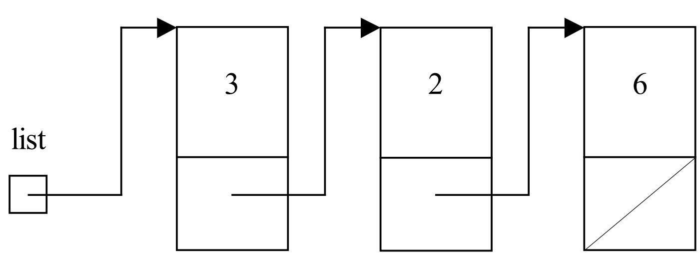

Figure 13 an example of a non-empty list

- several ways to do this. A good hint when removing something is:
  - "set a reference to item being removed"
- applying this convention gives something like:

r = list	- set r to item being removed
x = r.info	- get its info
list = r.next	- update the list
r.freeItem()	- don't forget to free memory we don't need!

  - (note that we can't retrieve from an empty list – would have to handle this by testing first for `isEmpty()`, which we'll do next)
- BTW, see that we just designed push() and pop()!

3. <u>Test for an empty list</u>

- this is easy! Draw the picture:

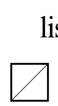

Figure 14 an empty list

- gives us the test that a list is empty when:

  list == null

4. <u>Can only access items in a single linked list sequentially – is called a traversal</u>

- e.g. count the number of items in this list:

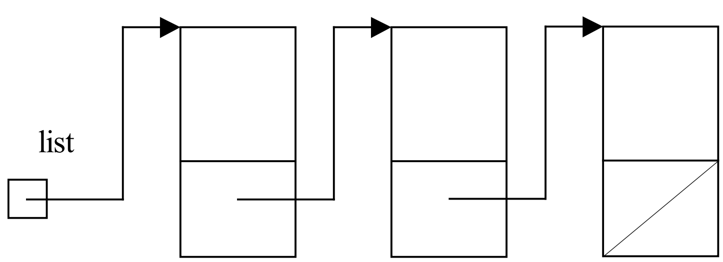

Figure 15 a non-empty list

  - HINT: list must always refer to the first item, we do not want to move it here. So use a 'working' reference to do the traversal. Call it r, for 'reference'

- traverse r from the beginning to the end of the list

```
count = 0
r = list
while (r != null)
    ++count
    r = r.next
print count
```

  - traversal is very common, we do it all the time with lists

  - (note that this works for an empty list also)

<u>Summary</u>

- introduced the single linked list abstraction, designed some list processing algorithms

- it is essential to use pictures to design algorithms

  - particularly as data structures will become more complex!

- use simple pseudocode to express the algorithms at first

  - but with more experience in your programming language, you will find that your pcode becomes Java

  - eventually you will go directly from designing an algorithm with pictures to implementing it in code

- will continue to design linked list algorithms using pictures and our pseudocode…

Insert and delete for a list
Objective: design these essential list operations

- unlike queues, we can insert and delete items from anywhere in a list, e.g.

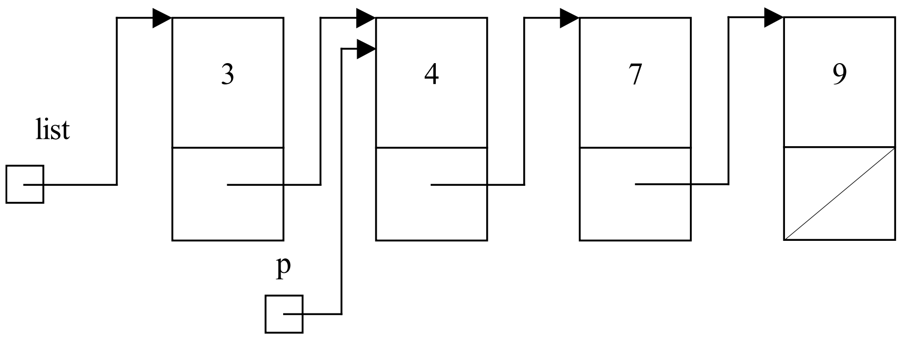

Figure 16 a reference p to "the item before the insertion or deletion point"

  - assume we have a reference p to "the item before the insertion or deletion point"
  - p will be set by searching for an item in the list, shortly
  - algorithms have to work for all cases, including the empty list special case

Design insertAfter() algorithm

- e.g. insert 6 after item pointed to by p
  - use the picture above to design the algorithm
  - we find two special cases. What if the list is empty? Or p does not refer to anything?
  - working with the picture, we come up with something like:

```
if (list is empty or p is not set)
   report error and exit
create a new item
initialize it
complete linking into list
```

- then more detailed, using our pseudocode:

```
if (isEmpty() || p == null)
   report error and exit
q = getItem()
```

g.info = 6
q.next = p.next
p.next = q

- note that we have to update the references in this order, otherwise the algorithm doesn't work

- see that there's no data movement – is very efficient!

<u>The deleteAfter() algorithm</u>

- delete the item <u>after</u> the item referenced by p. Return the deleted item's info

  - use the picture above

  - two special cases. What if p does not refer to anything? Or p refers to the last item in the list, so there is no item to delete?

  - then remember: when deleting an item – "set a reference to item being deleted"

  - so something like:

    if (p is not set or p is last item in list)
       report error and exit
    set reference q to item after p
    remove its information
    update p
    free old item q

- in pseudocode, something like:

```
if (p == null || p.next == null)
   report error and exit
q = p.next
x = q.info
p.next = q.next
q.freeItem()
```

  - note that there's a subtle problem here in the error testing, where we must test whether p refers to an item BEFORE we access its next value

  - more on this during implementation, next week

The find() algorithm

- searches the list to find first occurrence of an item. Returns a reference to the item, or null if not found
  - (so this is how we set the reference used by insert and delete)
  - e.g. find 4 in the starting list above
  - using the starting picture, we find just one special case, of an empty list
  - then come up with something like this. Is sequential or linear search. Start at the beginning and search every item in turn:

```
if (list is empty)
   report error and exit
start at first item in the list
while there are items remaining in the list and info is not 4
   move to next item
return reference to matching item or null
```

- in pseudocode, something like:

```
if (isEmpty())
   report error and exit
r = list
while (r != null && r.info != 4)
   r = r.next
return r
```

  - again, the same subtle problem here in the loop, where we must test whether r refers to an item BEFORE we access its info
  - problem is solved during implementation, next week

Summary

- update is easy and efficient when list is implemented dynamically
  - simply update one or two references!
- will implement these algorithms next week!

Next week and homework

- next week: implementation of linked lists

- homework by next week:

  - (no example programs this week)

  - read Horstmann, section 16.1

Lab

- lab assigned in ‘Queues’. See Canvas for due date