- dynamic implementation of single linked list
- double linked lists

Next week and homework

- next week: finish linked lists
- homework by next week:
  - download, run and understand this week's example programs
  - re-read Horstmann, section 16.1

Lab

- next lab is assigned. See Canvas for due date

Review last week

- introduced single linked lists last week
  - list consists of items, each with 2 instance variables:

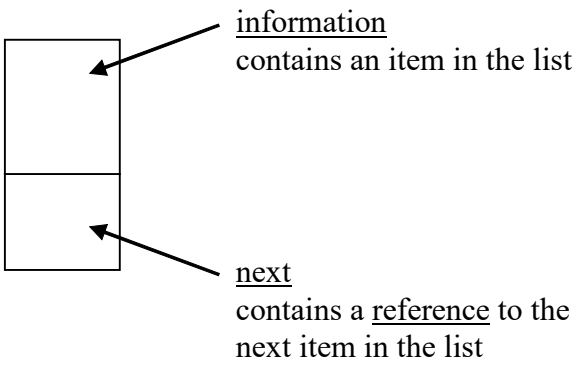

Figure 1 an item

  - items are connected together using references. Here's a single linked list:

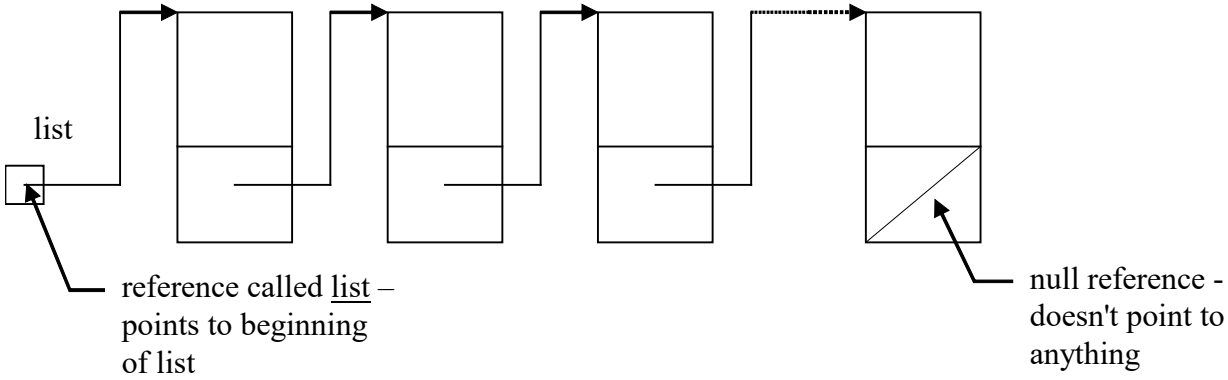

Figure 2 a single linked list:

- we introduced four primitives for working with items:

```
r.info
r.next
r = getItem()
r.freeItem()
```

- designed some list primitive algorithms by drawing pictures first, then pseudocode
  - essential to draw pictures to design complex algorithms, data structures!

Introduction to this week

- dynamic implementation of single linked lists
  - very similar to the pseudocode algorithms from last week!
- will implement the more powerful but more complex double linked list
- next lab, on lists, assigned this week

Dynamic implementation of single linked lists
Objective: finally get to implement lists dynamically in Java!

Did all the hard work last week!

- designed algorithms using pictures then pseudocode
  - easy to turn in to Java code now

Java does all memory management automatically and safely!

- last week we used getItem() and freeItem() primitives to do all memory management manually
  - getItem() – creates an item in memory
  - freeItem() – frees an item
- in languages like C, C++ it is the programmer's responsibility to manage memory, allocating and freeing memory as appropriate. Remember, in C:
  - `malloc()` a new item whenever we need one
  - `free()` an old item whenever we're finished with it
  - but allocating and freeing memory is notoriously difficult to do correctly, causes many problems
  - e.g. a memory leak is where a program continuously allocates memory without freeing it. Is bad programming, can cause the program to crash
- Java is deliberately designed to manage memory automatically and safely. Prevents a significant source of programming errors
  - getItem() to create an item is implemented by calling a constructor whenever necessary, no big deal e.g.

    ```
    Item r = new Item();
    ```

  - there is no such thing as free() in Java, to recover memory the programmer no longer needs. Instead this is done automatically by Java's automatic garbage collection mechanism

Implementation of single linked list

- e.g.

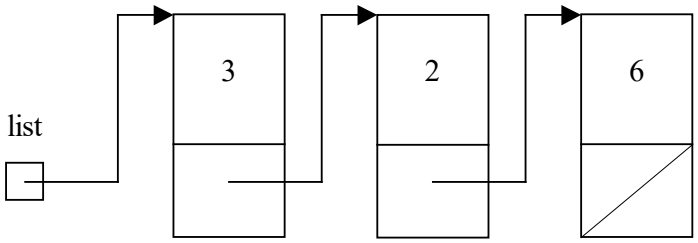

Figure 3 implementation of a single linked list:

  - maintain a reference named `list`, to the front of the list
  - unlike a queue, can insert, delete anywhere in a list
  - will only do a simple dynamic implementation of a list of `int`

- packaged of course into a `List` class
  - will use the familiar `Item` class, to give a list of items:

```
public class List
{
    private Item list;

    public List()
    {
        list = null;
    }
```

  - see how the constructor creates an empty list
  - now implement all the methods we designed last week

`insertFirst()` – insert a new item at the beginning of a list

- two different cases here, for an empty list and a !empty list
  - e.g. add item containing 1 to beginning of this list:

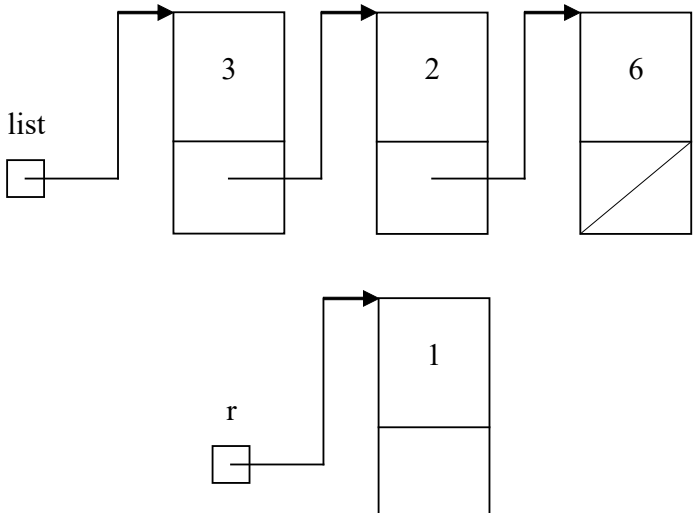

Figure 4 insertFirst() for a !empty list

– or to the beginning of the empty list:

Figure 5 an empty list

- using the pictures, we come up with an algorithm something like this, written as pseudocode:

  r = getItem() - create the new item
  r.info = 1 - set its info field
  r.next = list - link r into the beginning of the list
  list = r - update list

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

- now in Java:

```
public void insertFirst(int i)
{
    Item r = new Item(i);
    r.next = list;
    list = r;
}
```

`removeFirst()` – remove the first item from the beginning of a list and retrieve its information

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

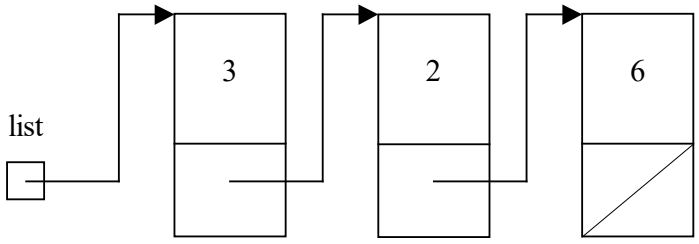

Figure 6 a !empty list

- remember: 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 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()`)
- now in Java:
  - implementations, so we have to handle all details
  - decide that we'll simply report an error and exit e.g. trying to remove from an empty list

```
public int removeFirst()
{
    if (isEmpty()) {
        System.out.println
          ("Error in removeFirst(): list is empty");
        System.err.println
          ("Error in removeFirst(): list is empty");
        System.exit(1);
    }
    Item r = list;
    int x = r.info;
```

```
        list = r.next;
        return x;
    }
```

- `System.exit()` causes the program to quit. It returns an integer value, used to report the reason for termination
- by convention, an exit value of 0 means there were no problems
- a non-zero value identifies what problem caused the exit

`isEmpty()` – tests for an empty list

- this is easy! Draw the picture:

Figure 7 an empty list

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

  list == null

- in Java, simply:

```
public Boolean isEmpty()
{
    return list == null;
}
```

`count()` – count the number of items in a list

- we can only access items in a single linked list sequentially – is called a traversal, e.g.

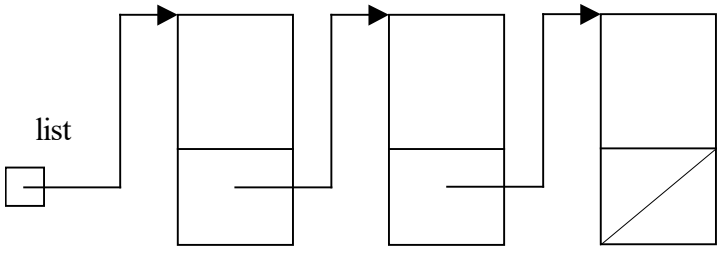

Figure 8 a linked list

- 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)

- in Java:

```
public int count()
{
    int count = 0;
    Item r = list;
    while (r != null) {
        ++count;
        r = r.next;
    }
    return count;
}
```

- BTW, C programmers would be tempted to rewrite the lengthy

```
while (r != null) {
```

  - as

```
while (r) {
```

  - however this is a syntax error in Java, due to the language's strict type checking. Error message is "incompatible types"

  - good, because the longer version is much more clear!

Insert and delete for a list

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

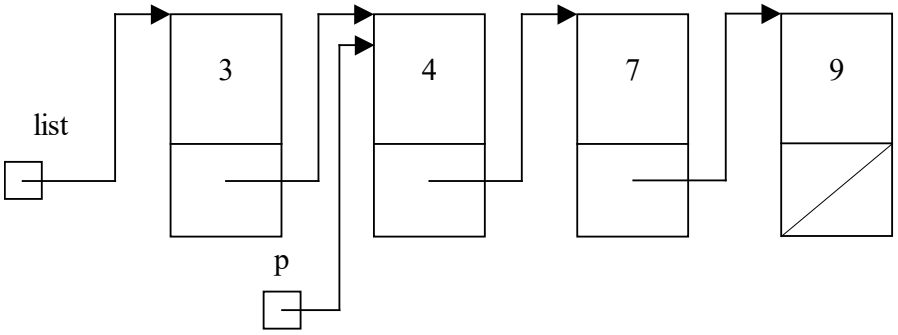

Figure 9 assume we have a reference p to "the item before the insertion or deletion point"

- 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

`insertAfter`() – insert after item pointed to by p

- e.g. insert 6 after item pointed to by p
  - working with the picture, we find two special cases. What if the list is empty? Or p does not refer to anything?
  - 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()       - create the new item
g.info = 6          - set its info field
q.next = p.next     - link q to next of p
p.next = q          - link p to q
```

- then in Java:

```
public void insertAfter(Item p, int i)
{
```

```
        if (isEmpty() || p == null) {
            System.out.println
             ("Error in insertAfter(): list is empty or p not set");
            System.err.println
             ("Error in insertAfter(): list is empty or p not set");
            System.exit(2);
        }
        Item r = new Item(i);
        r.next = p.next;
        p.next = r;
    }
```

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

`deleteAfter()` – delete the item after the item referenced by p

- returns the deleted item's info

  – 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 - q is next item
  x = q.info - get q's info
  p.next = q.next - link p to next of q
  q.freeItem() - free q

- in Java:

```
    public int deleteAfter(Item p)
    {
        if (p == null || p.next == null) {
            System.out.println
              ("Error in deleteAfter(): p not set or is last item");
```

```
            System.err.println
              ("Error in deleteAfter(): p not set or is last item");
            System.exit(3);
        }
        Item r = p.next;
        int x = r.info;
        p.next = r.next;
        return x;
    }
```

- note that there's a subtle implementation problem handled here. In the error testing, we are careful to test that `p` refers to an `Item` first, before getting its `next` value

- remember from C that evaluation of `&&`, `||` is guaranteed left to right. And that evaluation halts ('short circuits') as soon as the result is known

- so this code guarantees that we never try to dereference a null reference here

`find()` – returns reference to first occurrence of an item in the list, or null if not found

- will use sequential or linear search. Start at the beginning and search every item in turn

  - (so this is how we set the reference used by insert and delete)

  - e.g. find 4 in the starting list above

  - using the picture, we find just one special case, of an empty list

  - then come up with something like this:

    if (list is empty)
      report error and exit
    start at first item in the list
    while info is not 4 and there are items remaining in the list
      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
```

- in Java:

```
public Item find(int i)
{
    if (isEmpty()) {
        System.out.println("Error in find(): list is empty");
        System.err.println("Error in find(): list is empty");
        System.exit(4);
    }
    Item r = list;
    while (r != null && r.info != i)
        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 try to access its `info`

  - because of order of evaluation and short circuit, we are guaranteed never to dereference a null reference here

Review my `Simple list` example program

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

  - in `Tester::main()`, see how an example list is created

  - then tests insert and delete, count

  - then a traversal to print and dispose of the list

Summary

- see how implementation is very similar to the pseudocode algorithms from last week

  - so all the hard work is done during design, not during coding!

  - always draw pictures to design algorithms!

- Java deliberately takes responsibility for memory management

  - much safer than other languages, where programmers must control memory manually!

Double linked lists
Objective: single linked lists have a big restriction. Double linked lists avoid this shortcoming

Difficult to access the item before the current item

- in a single linked list, can only traverse from beginning to end, difficult to move in the other direction…

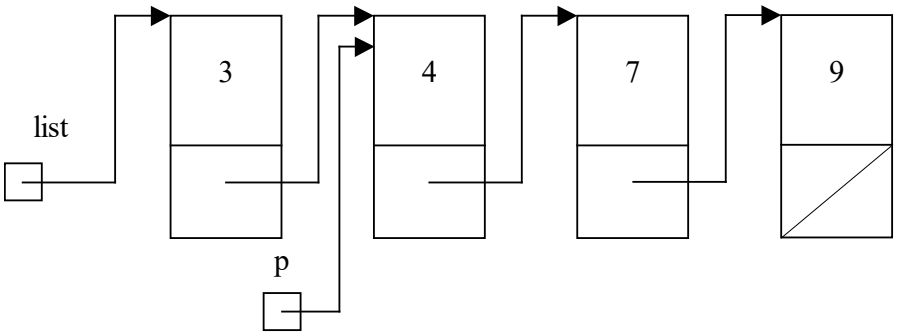

Figure 10 can only traverse a single linked list from beginning to end

  - …so it's difficult to do anything that requires us to update the item before the current item

  - e.g. to delete item pointed to by p

Can go backwards and forwards in a double linked list

- in a double linked list, each item has two references:

  - `next` – next item in list

  - `prev` – previous item in list

  - e.g.

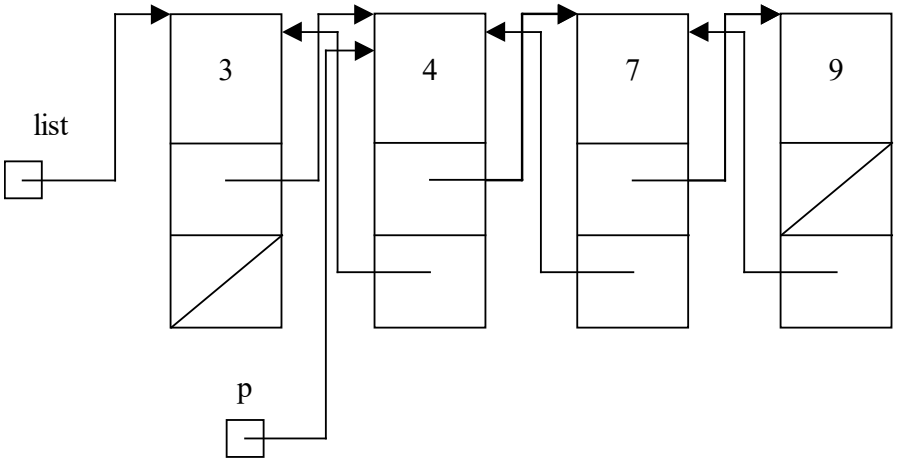

Figure 11 a double linked list

- so now can move from current item in either direction
- it's easy to update item before the current item

Design and implement a simple double linked list

- for practice. Will only do a very simple implementation
  - based around the previous simple linked list example program
  - will only cover here some of the most interesting double linked list methods

Need a new `DoubleItem` class

- need a new class for our items, that has the prev reference, e.g.

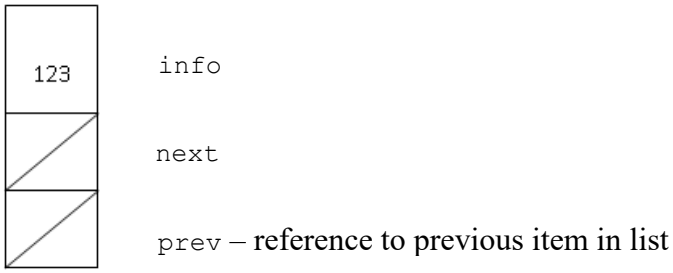

Figure 12 the new `DoubleItem` class

- implementation in Java is straightforward:

```
public class DoubleItem
{
    protected int info;
```

```
    protected DoubleItem next;
    protected DoubleItem prev;

    public DoubleItem()
    {
        info = 0;
        next = null;
        prev = null;
    }

    public DoubleItem(int i)
    {
        info = i;
        next = null;
        prev = null;
    }
}
```

- hint with `DoubleItem`: always be aware of and thinking about this additional `prev` reference and keeping it updated, as we'll see…

The new `DoubleList` class

- now a new class to use the new items, to give us a double linked list of items, as we've seen:

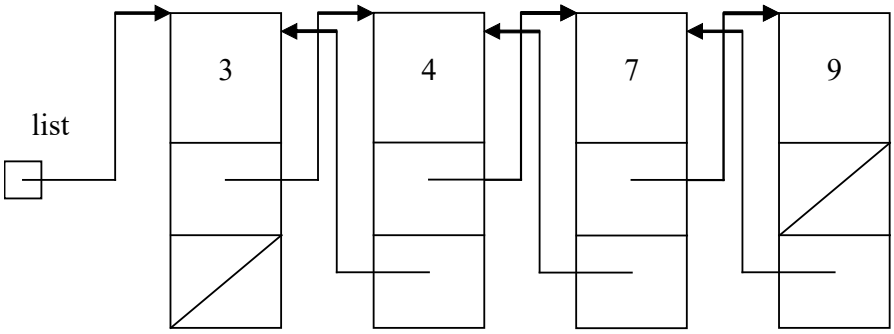

Figure 13 an example of a double linked list

  - see that we still maintain a reference named `list`, to the front of the list

- packaged of course into a `DoubleList` class

```
public class DoubleList
{
    private Item list;

    public DoubleList()
    {
        list = null;
    }
```

- see how the constructor creates an empty double linked list
- now design and implement three of the more interesting, challenging double linked list methods
- hint: always be thinking about keeping the `prev` reference updated!

1. `insertFirst()` – insert a new item at the beginning of a double linked list

- draw pictures, to identify two different cases here, for an empty list and a !empty list
  - 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 !empty double linked list:

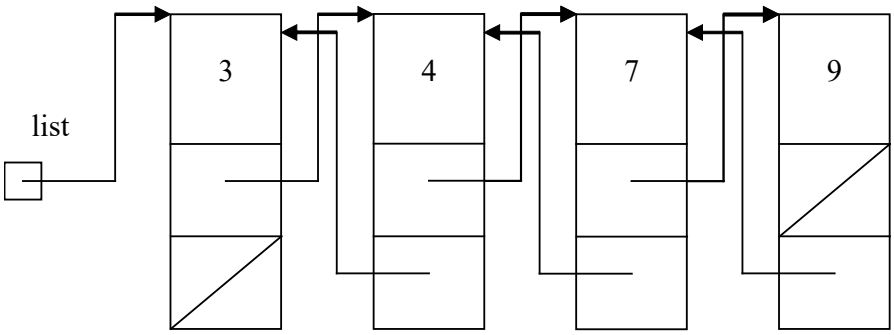

Figure 14 an example of a double linked list

  - or to the beginning of the empty list:

Figure 15 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 r into the beginning of the list
if (r.next != null)    - if there's a next item
   r.next.prev = r     - update its prev
list = r               - update list
```

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

- now turn the algorithm into Java. Something like:

```
public void insertFirst(int i)
{
    DoubleItem r = new DoubleItem(i);
    r.next = list;
    //if there's a next item, update its prev
    if (r.next != null)
        r.next.prev = r;
    list = r;
}
```

  - note first that the `DoubleItem()` constructor initializes both `next` and `prev` to `null`, so we do not have to update `r.prev` here
  - also, understand the new syntax `r.next.prev` here:
  - first, `r.next` is a reference to the next item in the list
  - then `prev` accesses its `prev` variable, exactly as we would expect
  - cool!

2. `delete()` – delete the item referenced by p. Return the deleted item's info

- hint: now that situations are becoming more complex, a good simplification is to create references to the left and right items of p. Draw the picture:

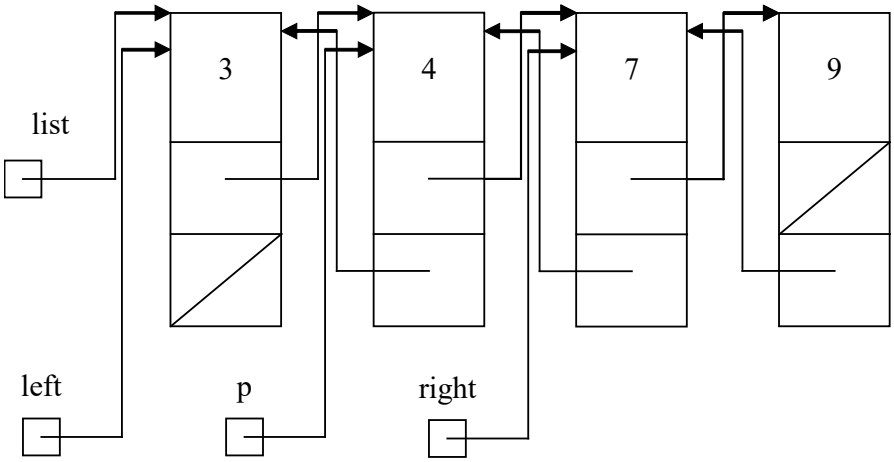

Figure 16 create references to the left and right items of p

  - left refers to previous item before p

- (can be null, if p is the first item in the list)
- right refers to next item after p
- (can be null, if p is the last item in the list)

- now use the picture to figure out steps to get to the desired end state. Something like:

```
x = p.info
set left references previous item
set right references next item
if (left == null)
   //we're deleting the first item from the list
   update list
else
   update left.next
if (right != null)
   //update the next item
   update right.prev
return x
```

- now turn the algorithm into Java. Something like:

```
public int delete(DoubleItem p)
{
    if (p == null) {
        System.out.println("Error in delete(): p not set");
        System.err.println("Error in delete(): p not set");
        System.exit(3);
    }
    int x = p.info;
    DoubleItem left = p.prev;    //left is previous item
    DoubleItem right = p.next;   //right is next item
    if (left == null)
        //we're deleting the first item from the list
        list = right;
    else
        left.next = right;
    if (right != null)
        //update the next item
        right.prev = left;
    return x;
}
```

- did some error checking in the implementation, that program reports error and exits if p is not set
- assumes that if p is set, then the list cannot be empty

3. `insertBefore()` – insert a new item before the item referenced by p

- in a double linked list can insert before or after the item referenced by p. Will only implement insert before in this example program, takes advantage of the prev reference

- for simplicity, will assume that we cannot insert into an empty list

- draw pictures, to identify two different cases here, for p is the first item or !first
  - then use the pictures to figure out steps to get to the desired end state
  - e.g. insert item containing 1 before p, where p is the first item:

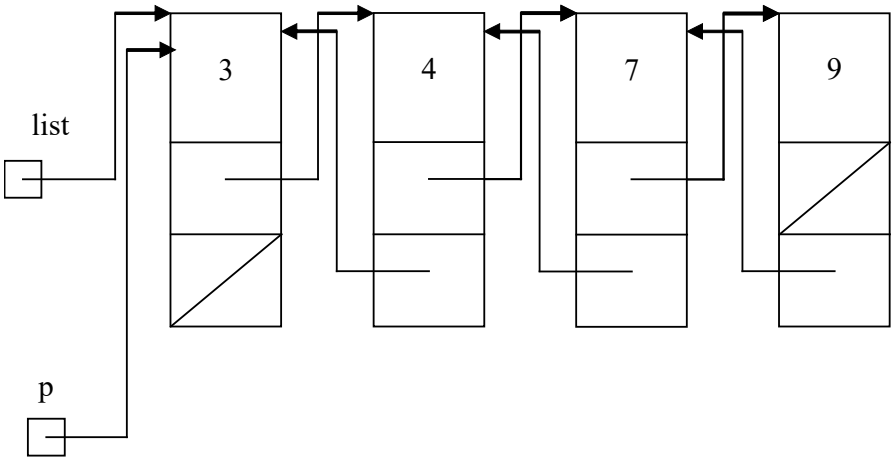

Figure 17 p is the first item

  - e.g. or where p is not the first item:

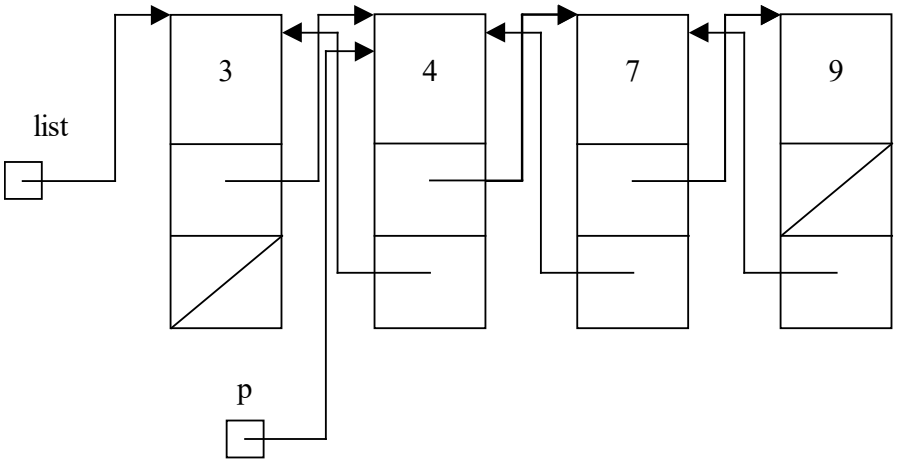

Figure 18 p is not the first item

- using the pictures, we come up with an algorithm something like this, written as pseudocode. Uses one reference:

  – left refers to previous item before p

```
r = getItem()       - create the new item
r.info = 1          - set its info field
left = p.prev       - left is previous item
p.prev = r          - link p back to r
r.next = p          - link r to p
r.prev = left       - link r back to left
if (left == null)   - if inserting before first
   list = r                 - update list
else                - else inserting between left and p
   left.next = r            - link left to r
```

- now turn the algorithm into Java. Something like:

```
public void insertBefore(DoubleItem p, int i)
{
    if (isEmpty() || p == null) {
        System.out.println
        ("Error in insertBefore(): list is empty or p not set");
        System.err.println
        ("Error in insertBefore(): list is empty or p not set");
        System.exit(2);
    }
    DoubleItem r = new DoubleItem(i);
    DoubleItem left = p.prev;   //left is previous item
    p.prev = r;
    r.next = p;
    r.prev = left;
    if (left == null)
        //then p is first item in list
        list = r;
    else
        left.next = r;
}
```

  – see here we decided to report an error and exit if we try to insert before a list that is empty, or if p is not set

Summary

- a list is often implemented as a double linked list, because then it's easy to work with the item referenced by a reference

  – slight overhead over a single linked implementation for the additional reference

  - methods are more complex

  - …but more convenient and cool!

- from Canvas, ‘Implementation of linked lists’ module, Example programs, download, read, run and understand my `Simple double linked list` example program

  - read and understand `Tester::main()`, see how it tests the double linked list in a similar way to the single linked list previously

  - understand the other methods not covered above

Next week and homework

- next week: finish linked lists
- homework by next week:
  - download, run and understand this week's example programs
  - re-read Horstmann, section 16.1

Lab

- next lab is assigned. See Canvas for due date