ECS510

Algorithms and Data Structures in an Object Oriented Framework “ADSOOF”

Linked Lists

Implementing Lisp lists

- We have already seen the importance of separating implementation from application of an abstract data type
- We have seen Lisp lists as an abstract data type, defined precisely by how the `head`, `tail`, `cons`, `empty` and `isEmpty` operations work
- We need to consider what could go inside a Lisp list object to make these methods work correctly

An Array Representation

- We could have inside a Lisp list object, an array in which the items of the Lisp list are stored in the same order in the array

```
class LispList<E>
{
 private E[] array;

 private LispList(E[] a)
 {
  array = a;
 }
…
```

Private constructor

- There is a `private` constructor for the class which takes the data structure representation and returns a new object with this data structure inside it - needed for constructive change
- The public methods which return new `LispList` objects construct the new data structure for the new object, then use the private constructor to create the new object which is returned
- Keeping the constructor private ensures the application code does not have access to the inside of Lisp lists, so they cannot be changed in unexpected ways

Lisp List methods in array implementation

```
public E head() {
 return array[0];
}

public LispList<E> tail() {
 E[] a = (E[]) new Object[array.length-1];
 for(int i=1; i<array.length; i++)
    a[i-1]=array[i];
 return new LispList<E>(a);
}

public LispList<E> cons(E obj) {
 E[] a = (E[]) new Object[array.length+1];
 a[0]=obj;
 for(int i=0; i<array.length; i++)
    a[i+1]=array[i];
 return new LispList<E>(a);
}
```

Generic Typing

- Lisp lists need a element type, but the class is generic, hence the type variable `E`
- The methods `cons` and `tail` return a new Lisp list of the same element type of the list they are called on
- `LispList.<Thing>empty()` creates a new Lisp list with element type `Thing`

Type variables in generic class

- In class `LispList<E>` the methods use the type variable `E`, meaning the element type of the Lisp list object they are called on
- Static method `empty` has its own type variable, as it is not called on an object:

```
public static <T> LispList<T> empty()
{
 return new LispList<T>((T[]) new Object[0]);
}
```

Linked Lists

- Linked lists are a data structure which resembles the Lisp list abstract data type

```
class Cell<T>
{
 T first;
 Cell<T> rest;

 Cell(T h, Cell<T> t)
 {
  first=h;
  rest=t;
 }
}
```

Class `Cell`

- A generic class
- Breaks the rule that variables inside a class should be `private` or `protected`
- Recursive - a `Cell` object contains a reference to a `Cell` object
- But the reference can be set to `null`
- A simple use of Java techniques for illustration, not a built-in part of Java

Data structure v. ADT

- An Abstract Data Type is seen only in terms of the methods it has, they define it completely
- A linked list is a data structure, because we consider it in terms of actual variables, `first` and `rest` which can be accessed directly
- A linked list consists of a data item (`first`) and a further linked list (`rest`)
- This corresponds closely to the head and tail of a Lisp list

Implementing Lisp list with linked list

- Use the same principle as previously
  - A variable holding the data structure
  - A private constructor which takes a data structure argument and returns a new object
  - The methods work by creating a new data structure representing the object required, then using the private constructor to create the object required

```
class LispList<E>
{
 private Cell<E> myList;

 private LispList(Cell<E> list)
 {
  myList=list;
 }

...
```

Nested class `Cell`

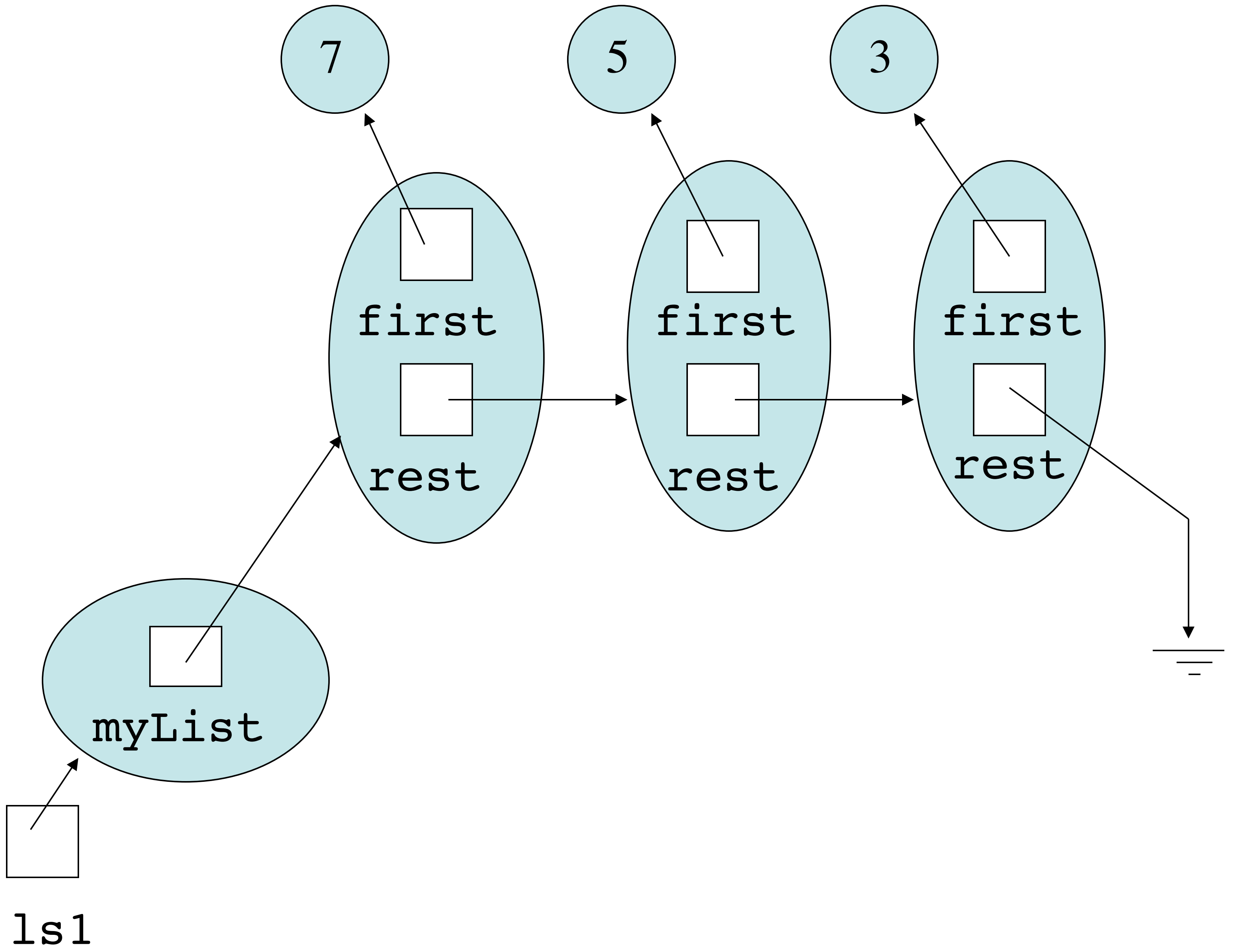
- A nested class is a class declared inside another class
- Declaring `Cell` as a `private` nested class in `LispList` is a way of making it for the internal use of `LispList` methods only
- As only internal code can access it, unprotected access to a `Cell` object's variables is acceptable
- As `Cell` is declared as `static` it is self-contained (non-static nested classes are known as "inner classes" and raise further issues not considered here)

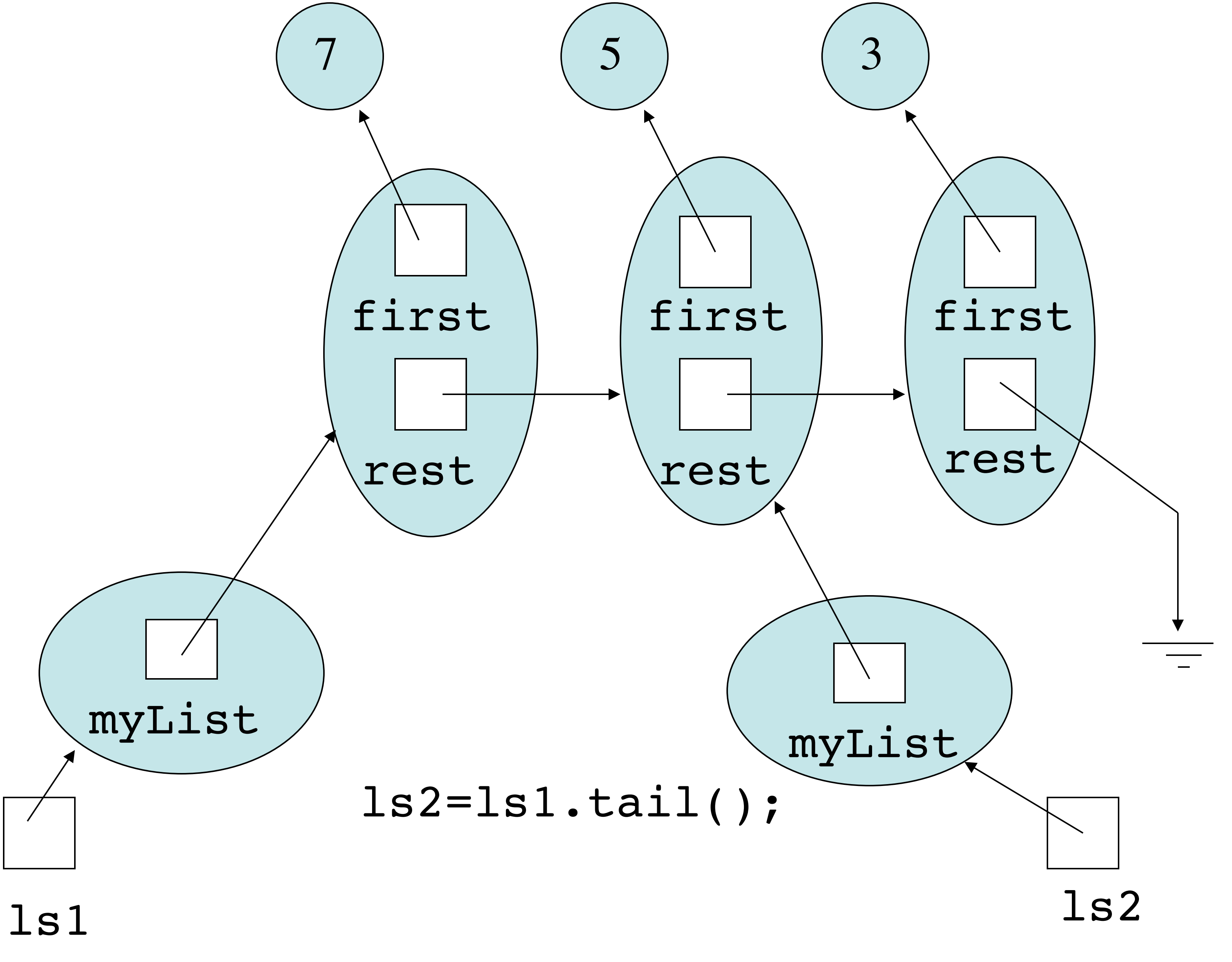
Cell representation of Lisp lists

- Represent an empty Lisp list by a `LispList<T>` object whose `myList` variable is set to `null`
- Otherwise, represent a Lisp list by a `LispList<T>` object whose `myList` variable refers to a `Cell<T>` object whose `first` variable is the Lisp list's head, and whose `rest` variable is the linked list data structure for its tail
- Keep remembering the distinction between `LispList` (abstract data type) and `Cell` used to make data structures <u>inside</u> a `LispList` object

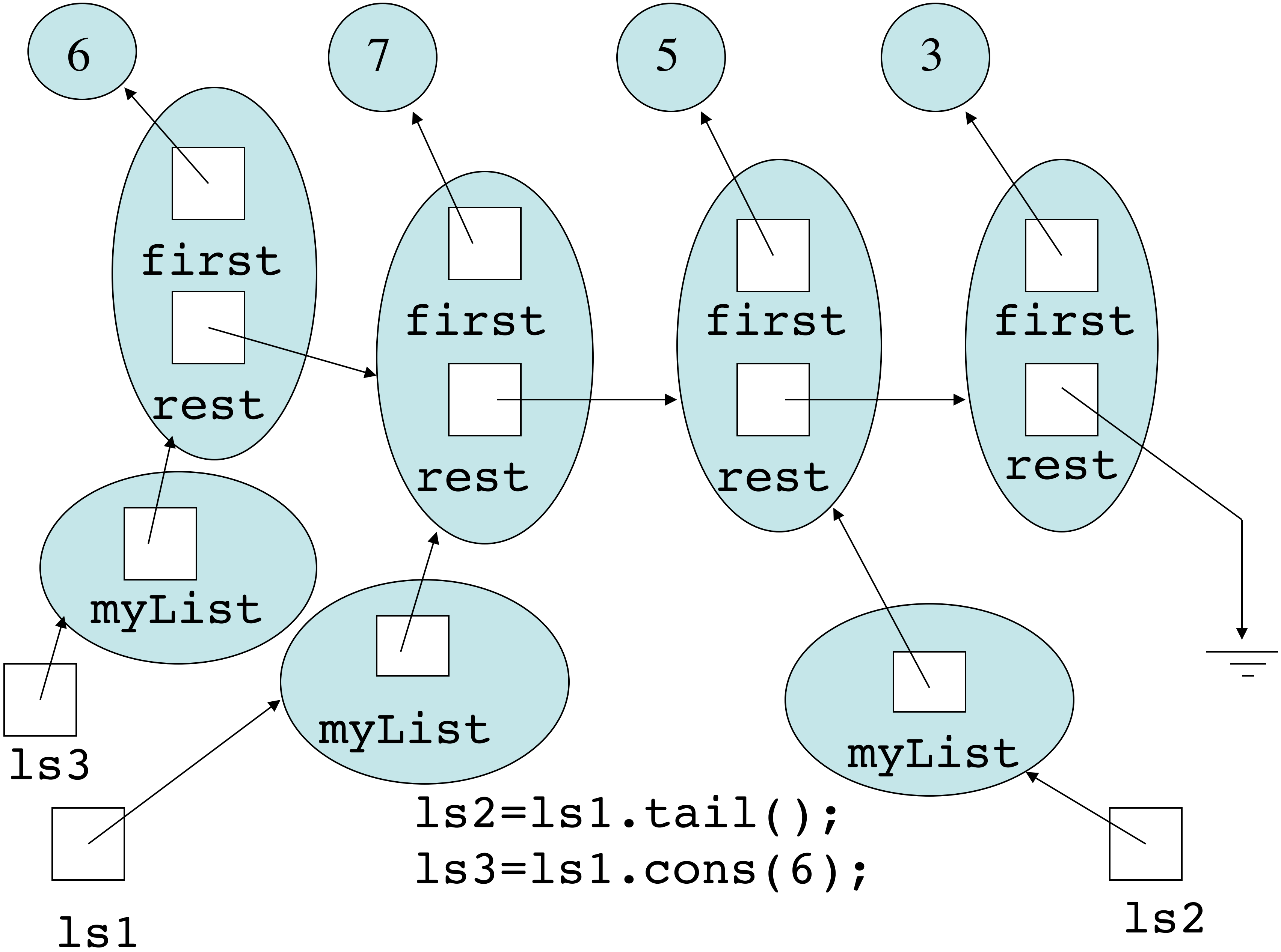
Example

- We think of a particular `LispList<Integer>` object as representing [7,5,3] - a Lisp list whose head is 7 and whose tail is the Lisp list [5,3]
- The text representation of this is the `String` `"[7,5,3]"`
- We don’t know what is actually inside the `LispList` object, could be an array, could be a linked list structure
- So long as `LispList` is properly implemented, it does not matter what is inside, and it cannot affect how the application code works logically

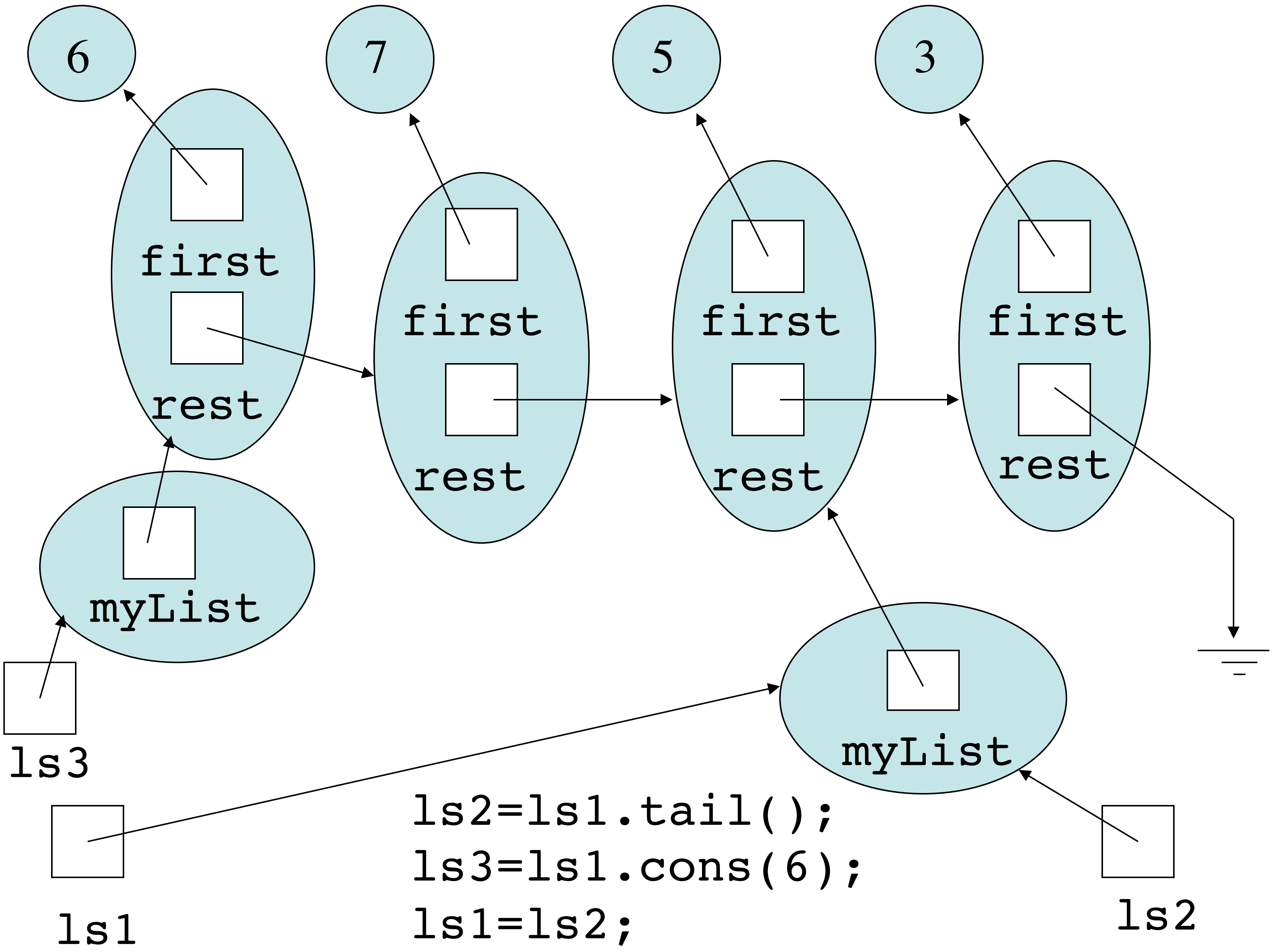

7
5
3
first
rest
first
rest
first
rest
myList
ls1



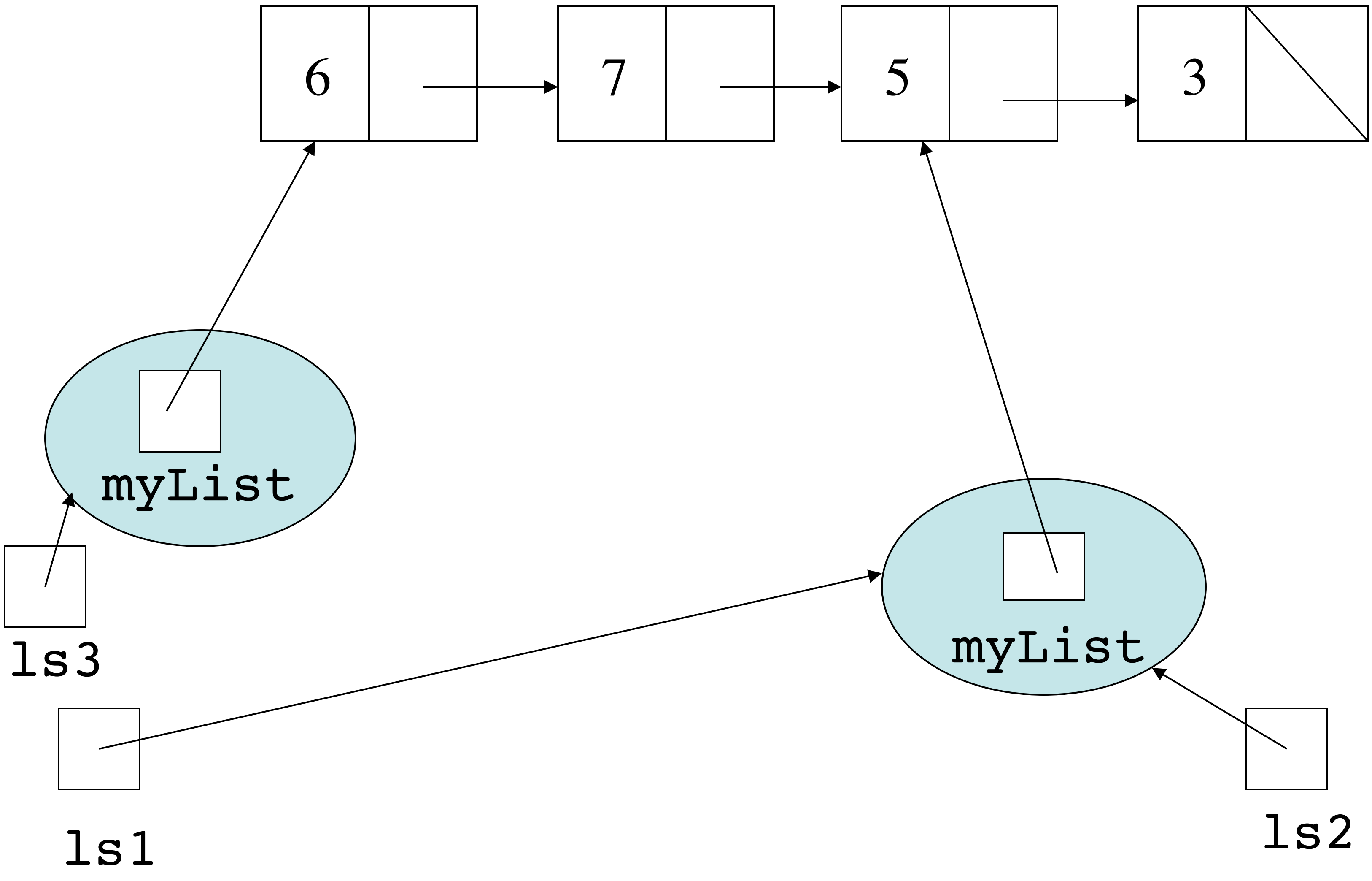
`ls2=ls1.tail();`




```
ls2=ls1.tail();
ls3=ls1.cons(6);
```




```
ls2=ls1.tail();
ls3=ls1.cons(6);
ls1=ls2;
```


6
7
5
3
myList
myList
ls3
ls1
ls2

```
public E head() {
 return myList.first;
}

public LispList<E> tail() {
 return new LispList<E>(myList.rest);
}

public LispList<E> cons(E item) {
 return new LispList<E>(new Cell<E>(item,myList));
}

public static <T> LispList<T> empty() {
 return new LispList<T>(null);
}

public boolean isEmpty() {
 return myList==null;
}
```

Immutable objects

- Remember, Lisp lists as we have defined them are immutable
- That means there is no method you can call on a LispList object which changes it
- That is why they can be implemented using shared cells - there is no method you can call on a Lisp list object which would change the value of a cell it shares with another Lisp list object.

ArrayList implemented with a linked list

- The linked list data structure can be used to make a class of objects which behave like ArrayList objects (the code uses `next` instead of `rest`, but that's just a name change)
- The operations on the data structure to implement the ArrayList behaviour are more complex
- They include operations which change the data structure destructively
- The only constructor creates a new object representing an empty list

```
class MyArrayList<E>
{
 private Cell<E> myList;

 public MyArrayList()
 {
  myList=null;
 }

...
```

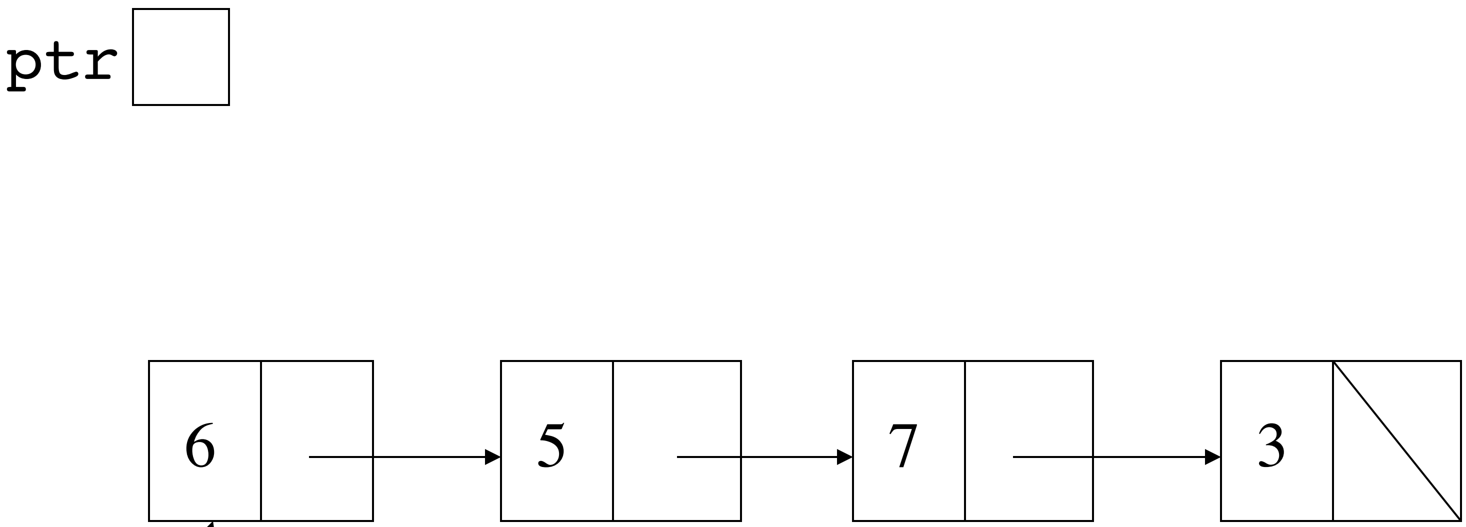
Some example methods

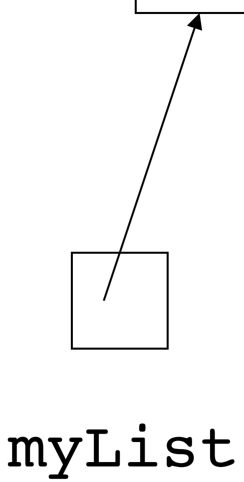
```
public E get(int pos)
{
 Cell<E> ptr=myList;
 for(int count=0; count<pos&&ptr!=null; ptr=ptr.next, count++)
  {}
 if(ptr==null)
    throw new IndexOutOfBoundsException();
 return ptr.first;
}

public void set(int pos,E item)
{
 Cell<E> ptr=myList;
 for(int count=0; count<pos&&ptr!=null; ptr=ptr.next, count++)
  {}
 if(ptr==null)
    throw new IndexOutOfBoundsException();
 ptr.first=item;
}
```

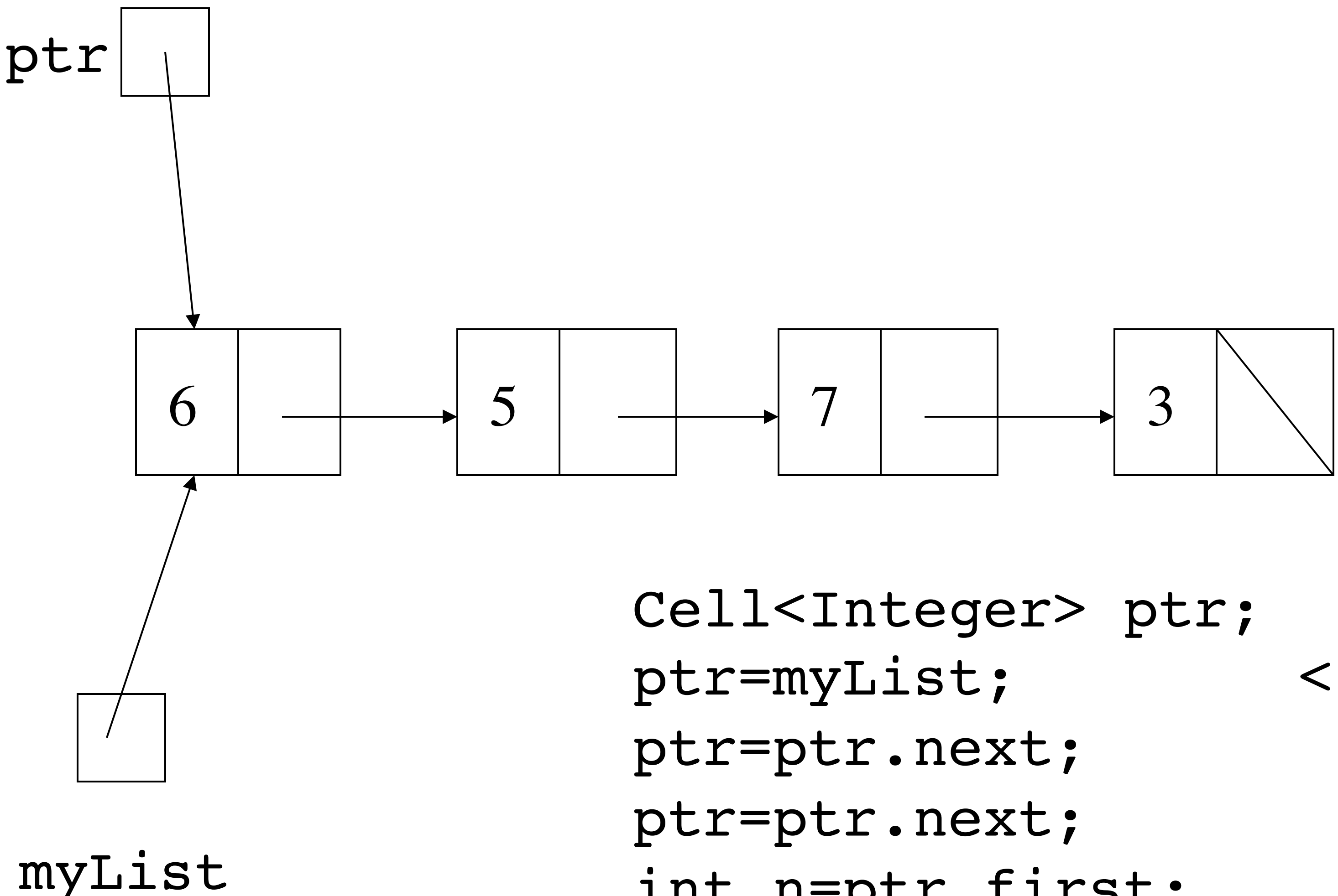
Moving pointer down a list

- Common way of manipulating linked lists
- Have variable of type `Cell`, often named `ptr`
- `ptr=list` sets `ptr` to first cell of list
- `ptr=ptr.next` moves `ptr` to next cell of list
- For diagrams, we will assume the element type is `Integer`

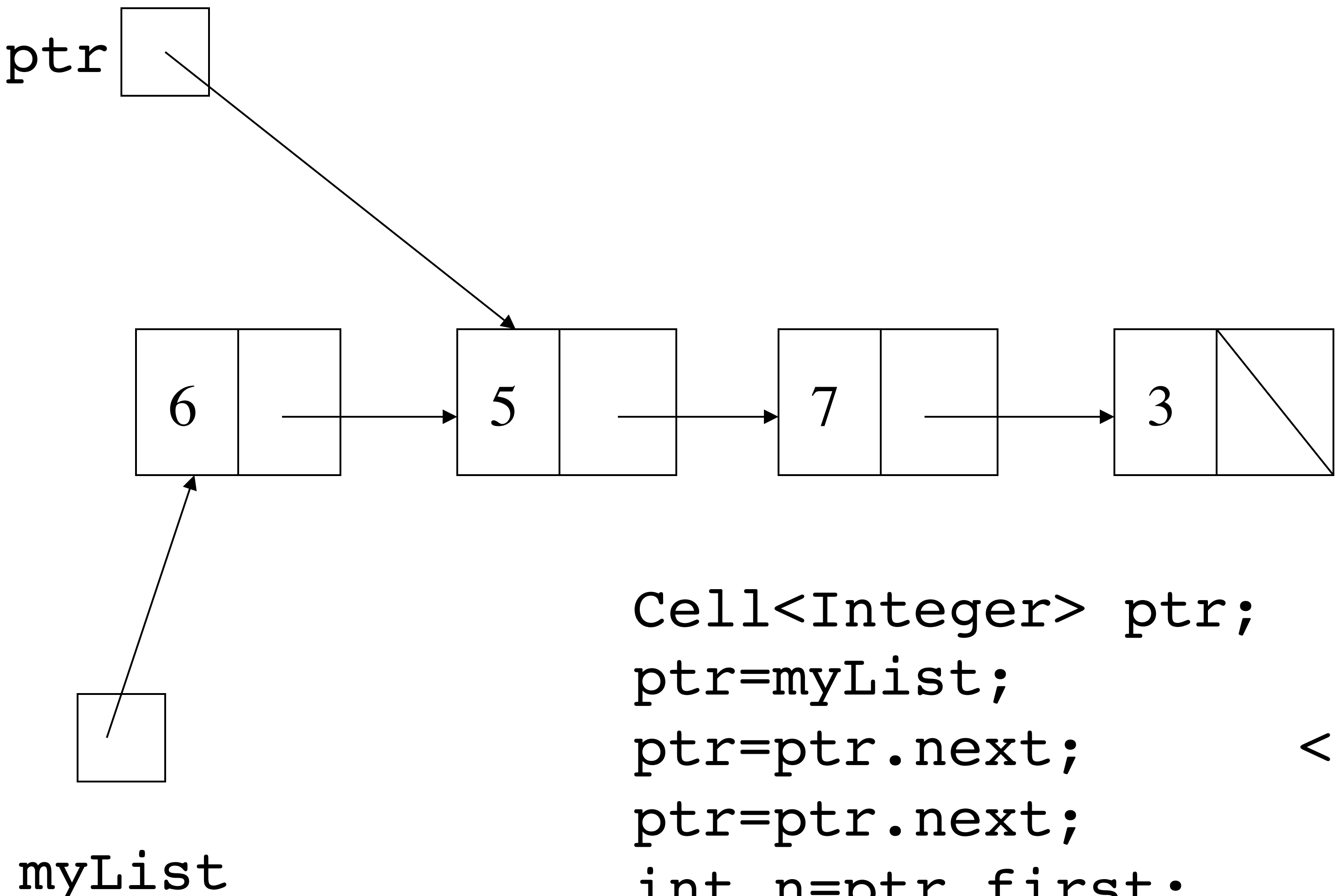




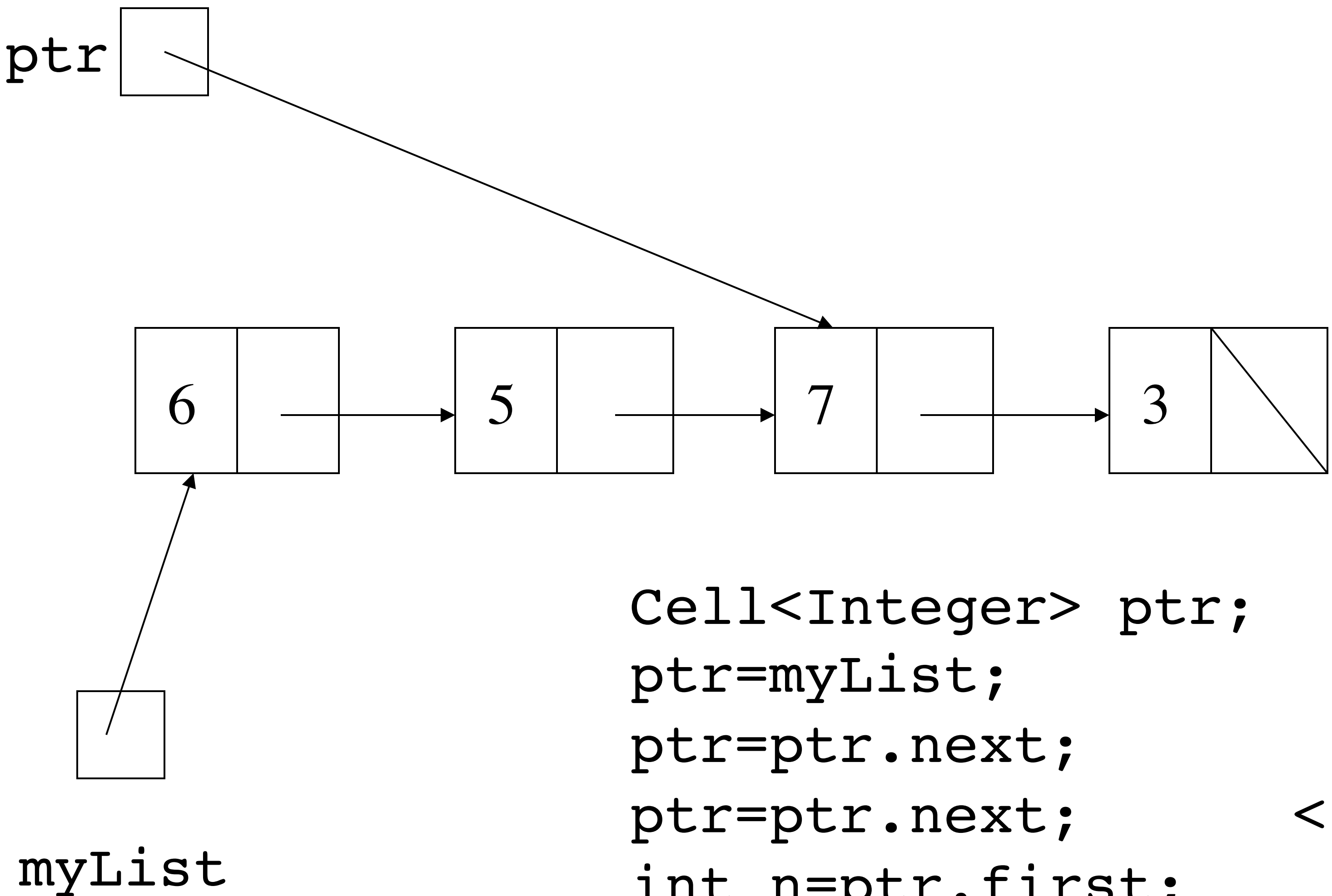

```
Cell<Integer> ptr; <
ptr=myList;
ptr=ptr.next;
ptr=ptr.next;
int n=ptr.first;
ptr.first=8;
```



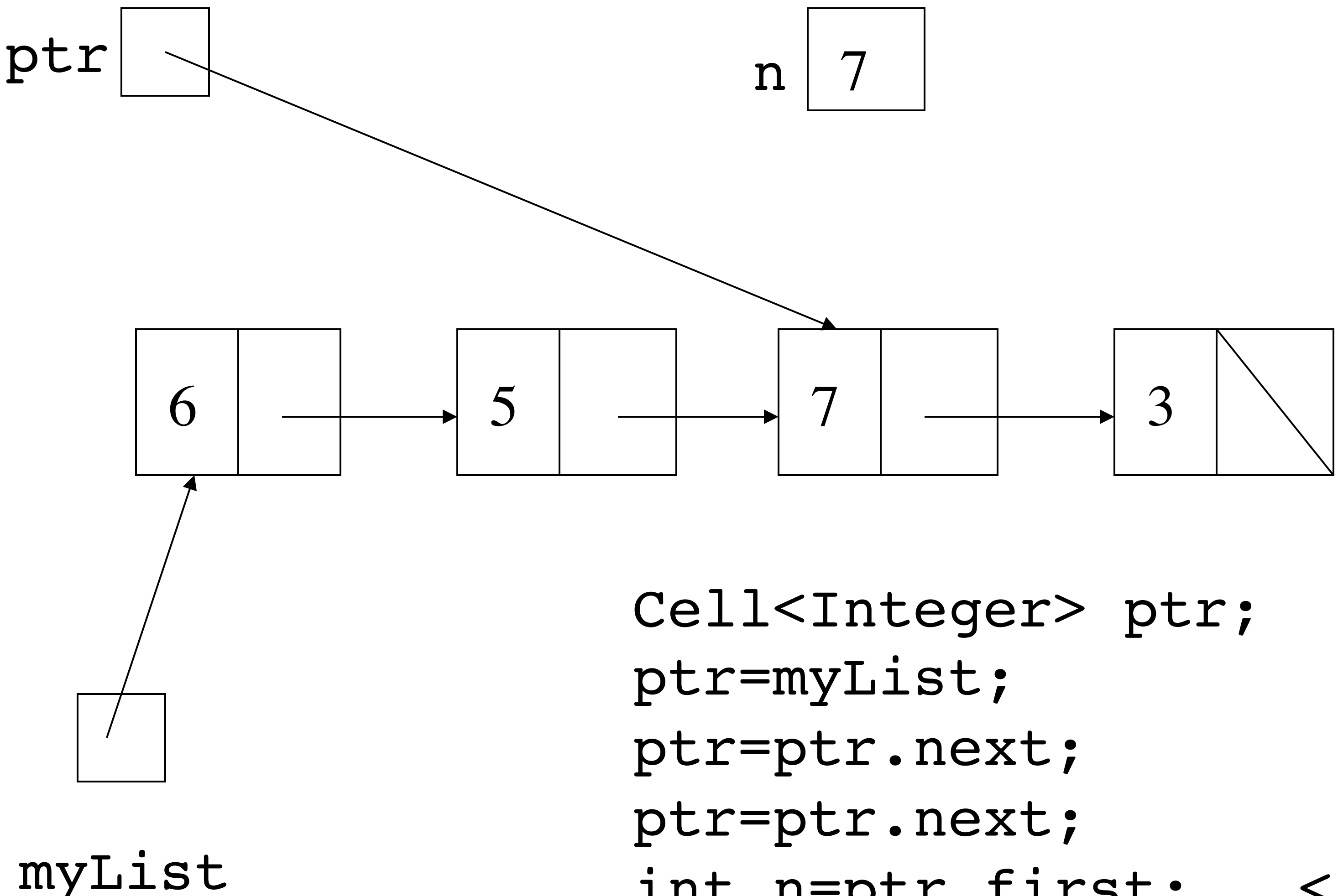

```
Cell<Integer> ptr;
ptr=myList;            <
ptr=ptr.next;
ptr=ptr.next;
int n=ptr.first;
ptr.first=8;
```



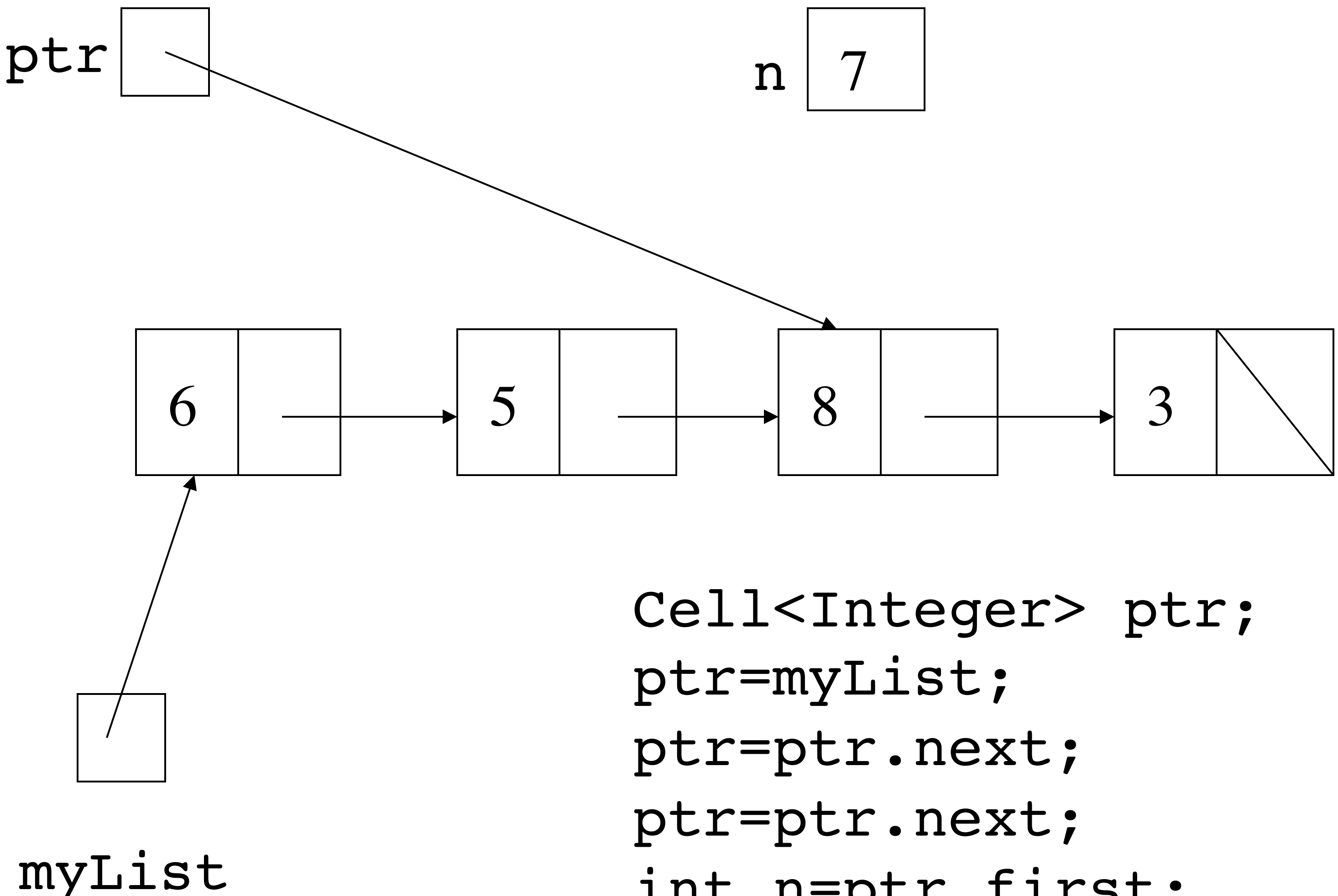

```
Cell<Integer> ptr;
ptr=myList;
ptr=ptr.next;          <
ptr=ptr.next;
int n=ptr.first;
ptr.first=8;
```




```
Cell<Integer> ptr;
ptr=myList;
ptr=ptr.next;
ptr=ptr.next;       <
int n=ptr.first;
ptr.first=8;
```




```
Cell<Integer> ptr;
ptr=myList;
ptr=ptr.next;
ptr=ptr.next;
int n=ptr.first;   <
ptr.first=8;
```



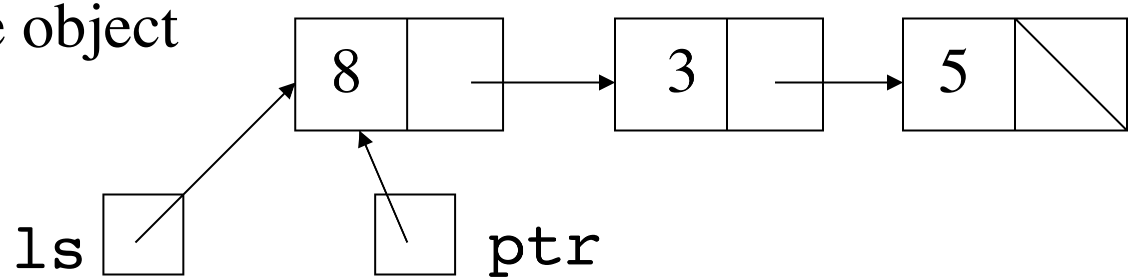

```
Cell<Integer> ptr;
ptr=myList;
ptr=ptr.next;
ptr=ptr.next;
int n=ptr.first;
ptr.first=8;        <
```

Cell and Pointer loops

- `for(ptr=ls; ptr!=null; ptr=ptr.next)`
  moves `ptr` all through list (assuming body has no `breaks`)
- ```
  for(int count=0, ptr=ls;
    count<pos&&ptr!=null;
    ptr=ptr.next, count++)
  ```
  moves `ptr` down list `pos` times, but ends with `ptr` set to `null` if `pos` is more than the number of cells `ls` has
- ```
  for(ptr=ls;
     ptr!=null&&!ptr.first.equals(n);
     ptr=ptr.next)
  ```
  moves `ptr` until it points to a cell containing `n`, or it has gone through all cells, in which case loop ends with `ptr` set to `null`

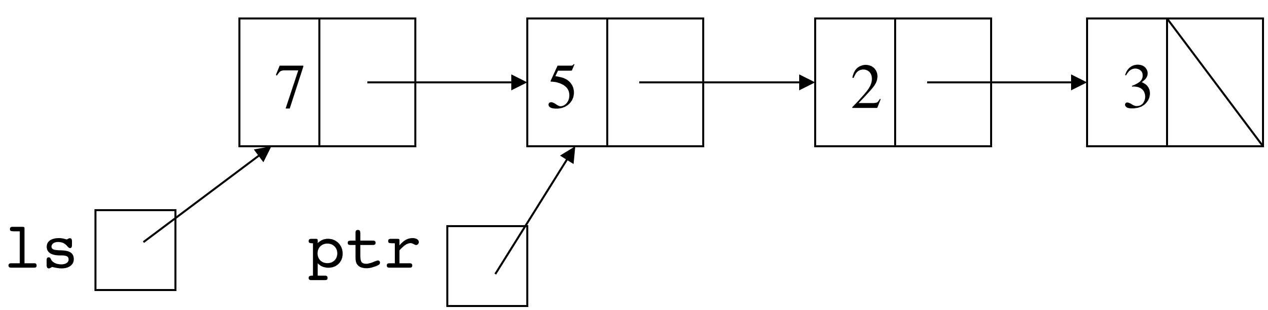
Aliasing in linked lists

- If `ptr` is of type `Cell<E>`, then `ptr` refers to an object which contains a variable `first` of type `E`, and a variable `next` of type `Cell<E>`
- The object may be aliased, so after `ptr=ls` is executed `ptr.first` and `ls.first` are two names for the same variable, and `ptr.next` and `ls.next` are two names for the same variable
- But `ptr` and `ls` are two separate variables referring to the same object



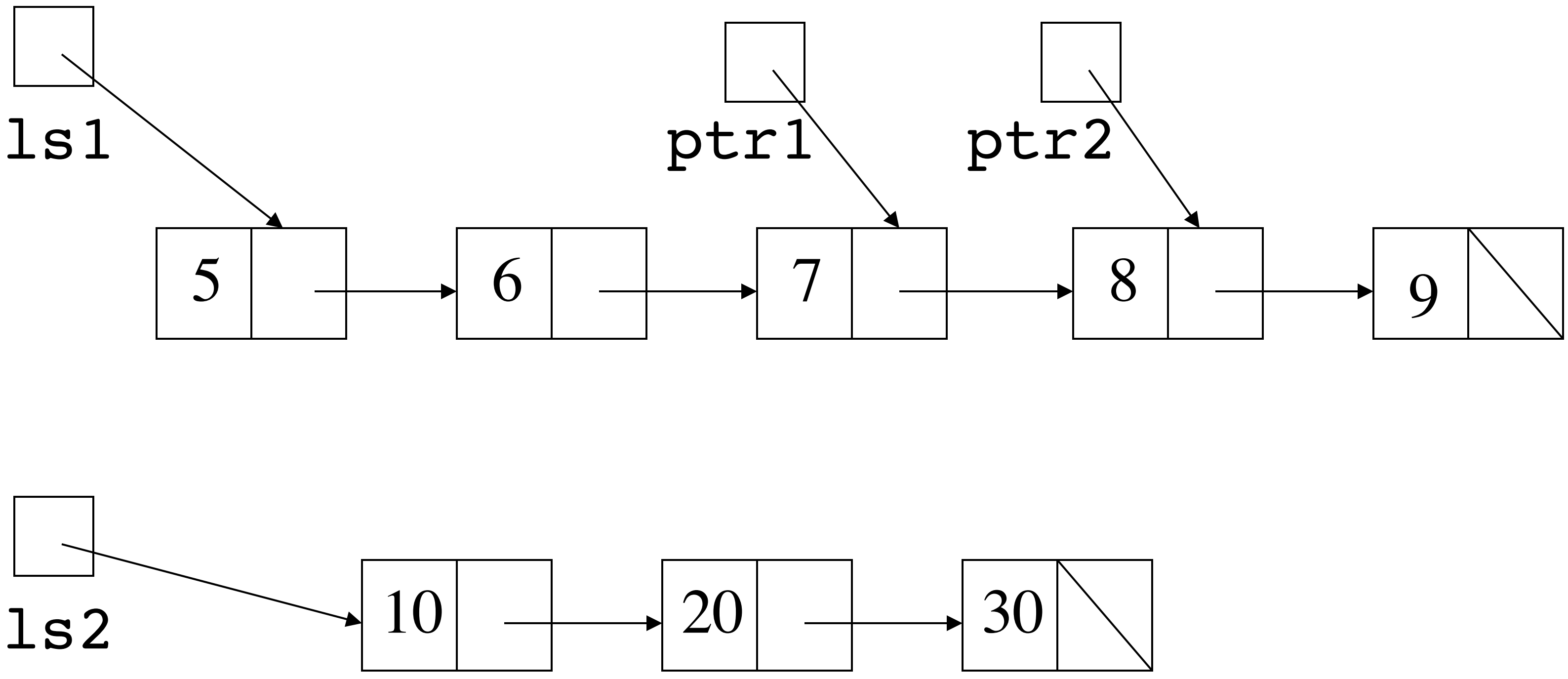
Pointing to a Cell in a structure

- Executing `ptr=ls.next` means `ptr` and `ls.next` are two <u>separate variables</u> referring to the same object
- Then also `ptr.first` is the <u>same variable</u> as `ls.next.first` and `ptr.next` is the same variable as `ls.next.next`

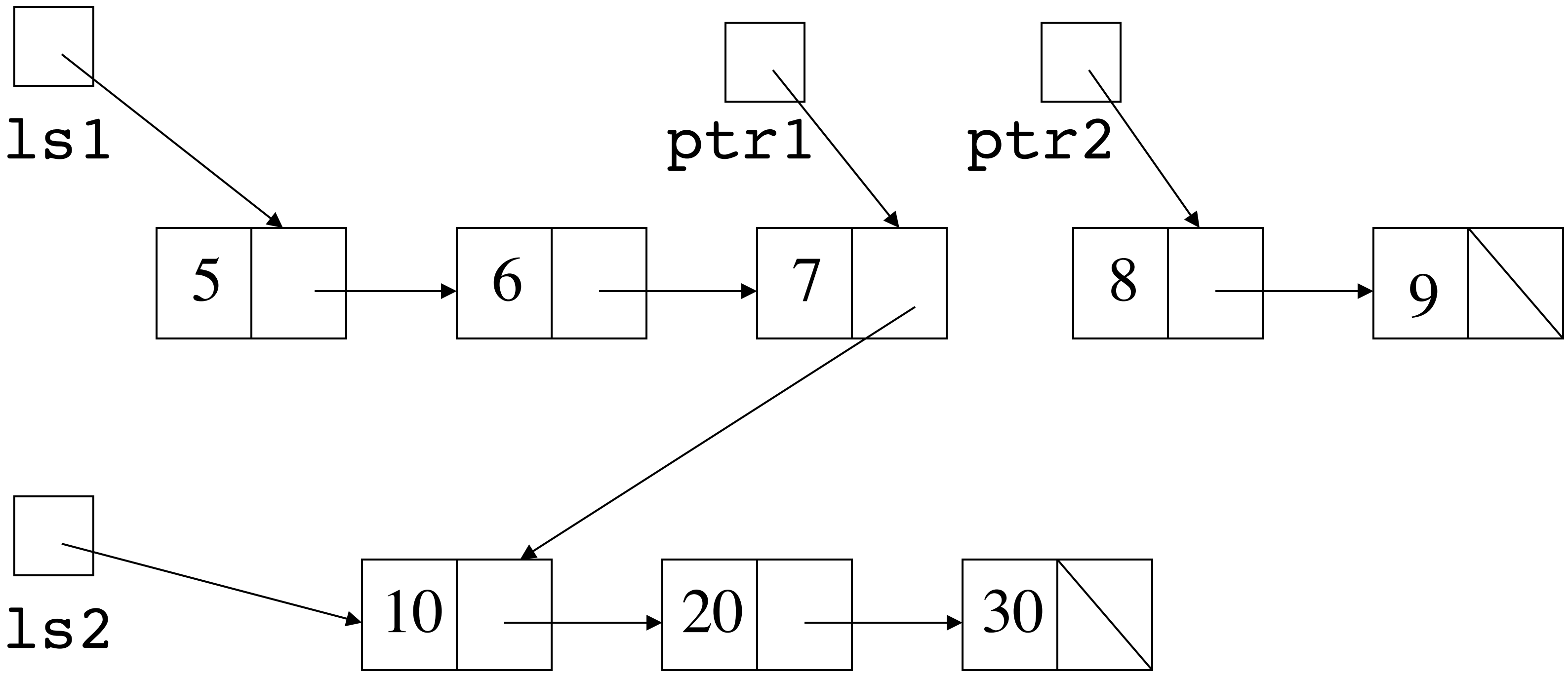


Destructive change in linked lists

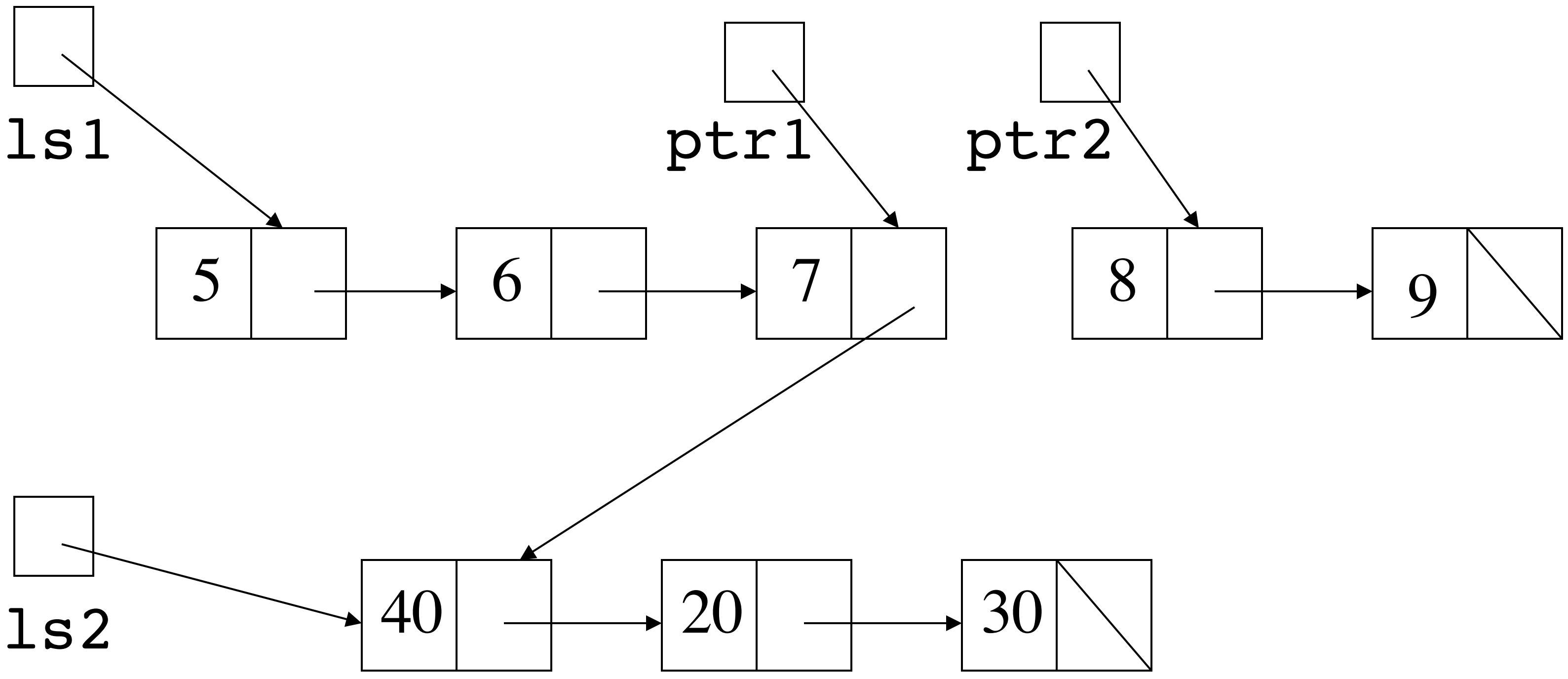
- `ptr.first=n` changes the value of the `first` variable of the `Cell` object it refers to
- `ptr.next=ls` changes the value of the `next` variable of the `Cell` object it refers to (remember object assignment is aliasing)
- If `ptr` refers to a `Cell` object which is in a linked list structure, the structure will be changed destructively
- This works because `first` and `next` are variables
- Compare with `LispList`, where the first item is accessed by the method call `head()` and the rest by the method call `tail()`
- You cannot assign to a method call `lsp.head()=8` ✗



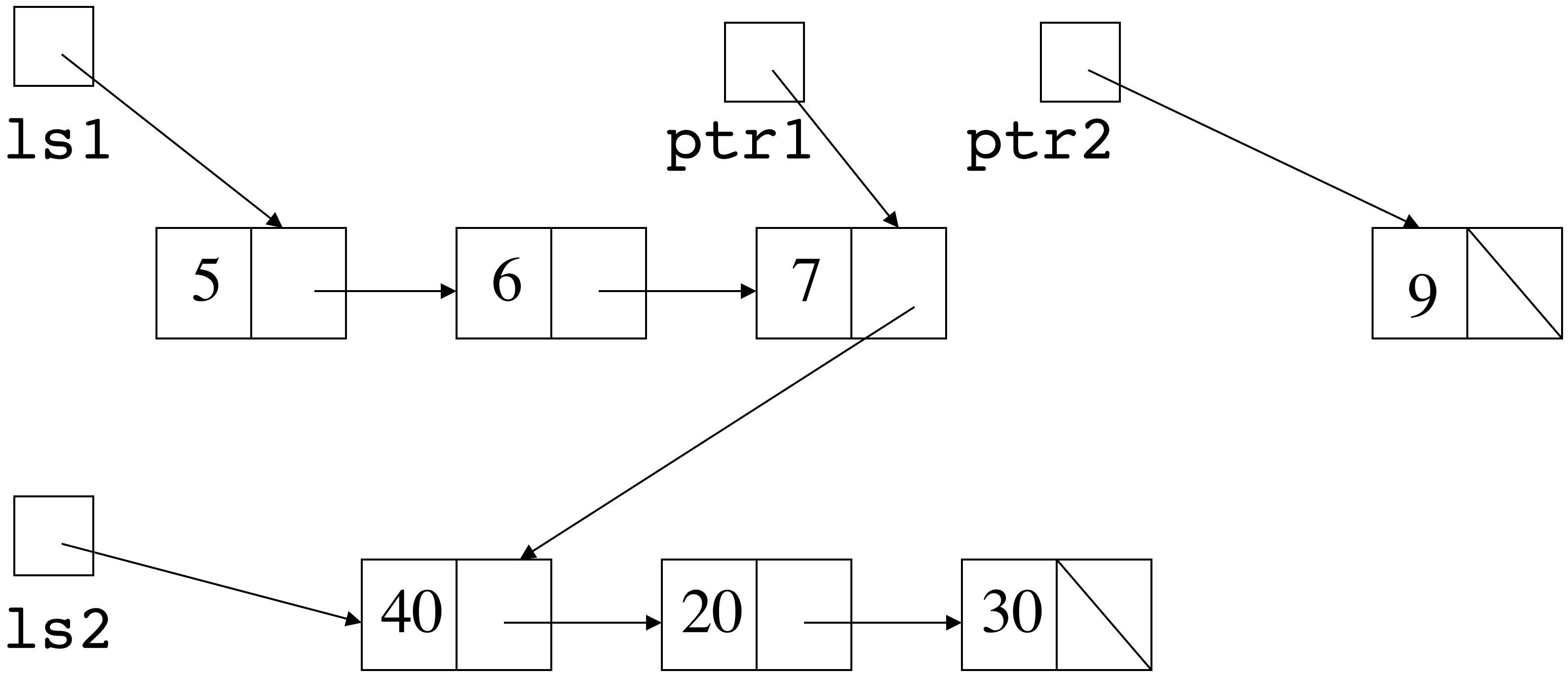

```
ptr1.next=ls2;
ptr1.next.first=40;
ptr2=ptr2.next;
ls2=new Cell(4,ls1);
ls1=ptr2;
```



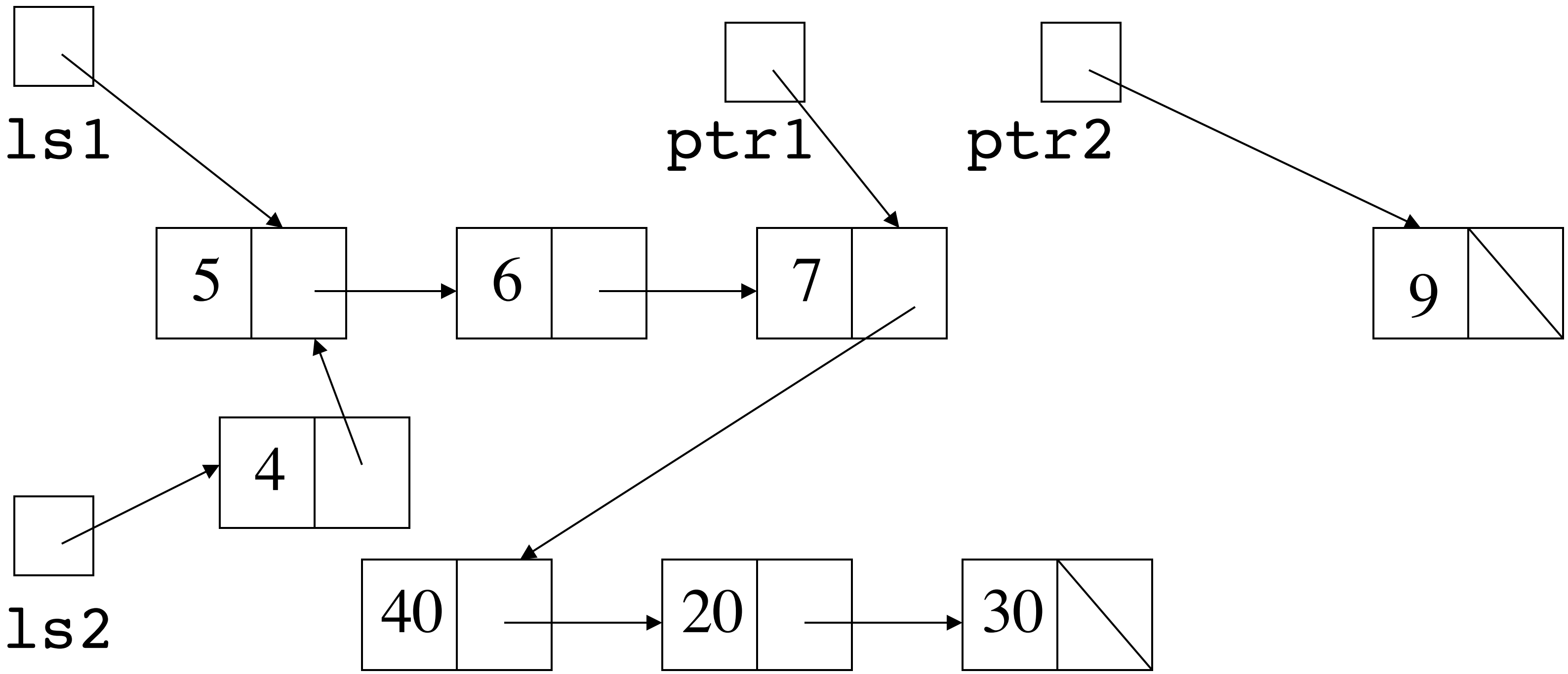

```
ptr1.next=ls2; <
ptr1.next.first=40;
ptr2=ptr2.next;
ls2=new Cell(4,ls1);
ls1=ptr2;
```



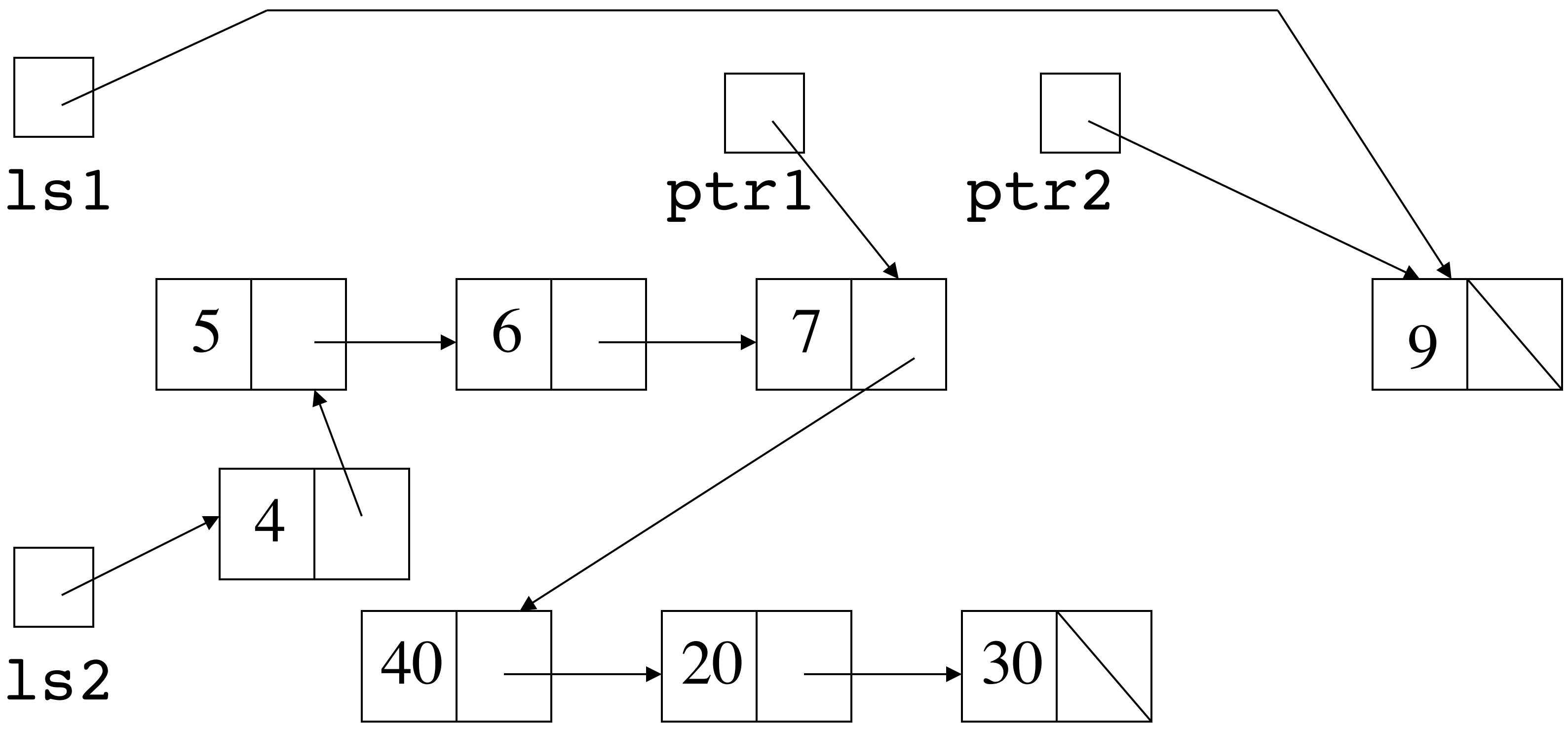

```
ptr1.next=ls2;
ptr1.next.first=40; <
ptr2=ptr2.next;
ls2=new Cell(4,ls1);
ls1=ptr2;
```




```
ptr1.next=ls2;
ptr1.next.first=40;
ptr2=ptr2.next; <
ls2=new Cell(4,ls1);
ls1=ptr2;
```



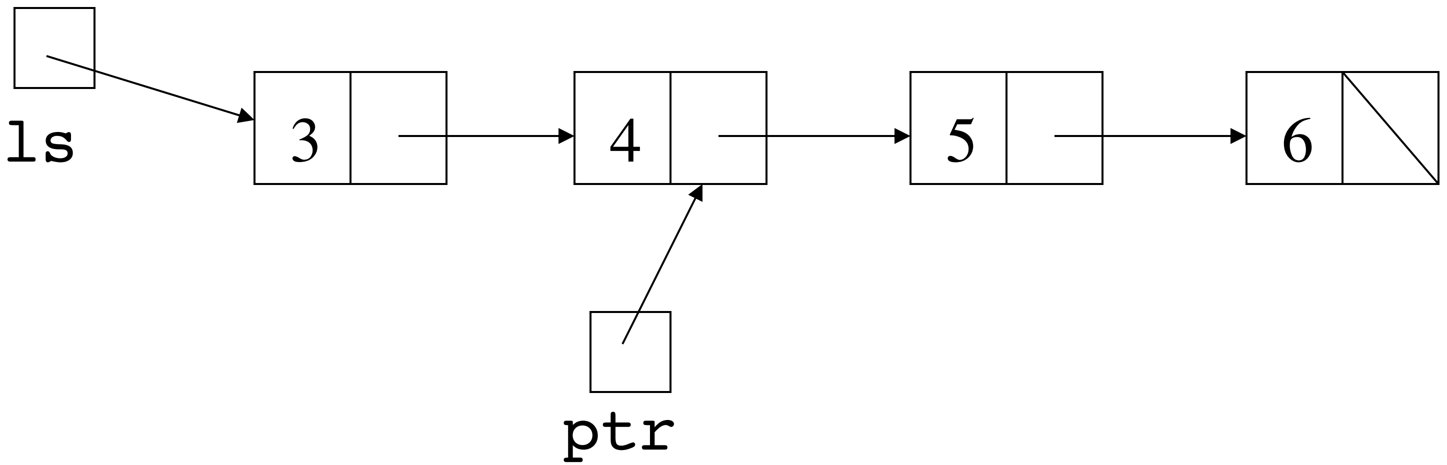

```
ptr1.next=ls2;
ptr1.next.first=40;
ptr2=ptr2.next;
ls2=new Cell(4,ls1); <
ls1=ptr2;
```




```
ptr1.next=ls2;
ptr1.next.first=40;
ptr2=ptr2.next;
ls2=new Cell(4,ls1);
ls1=ptr2; <
```

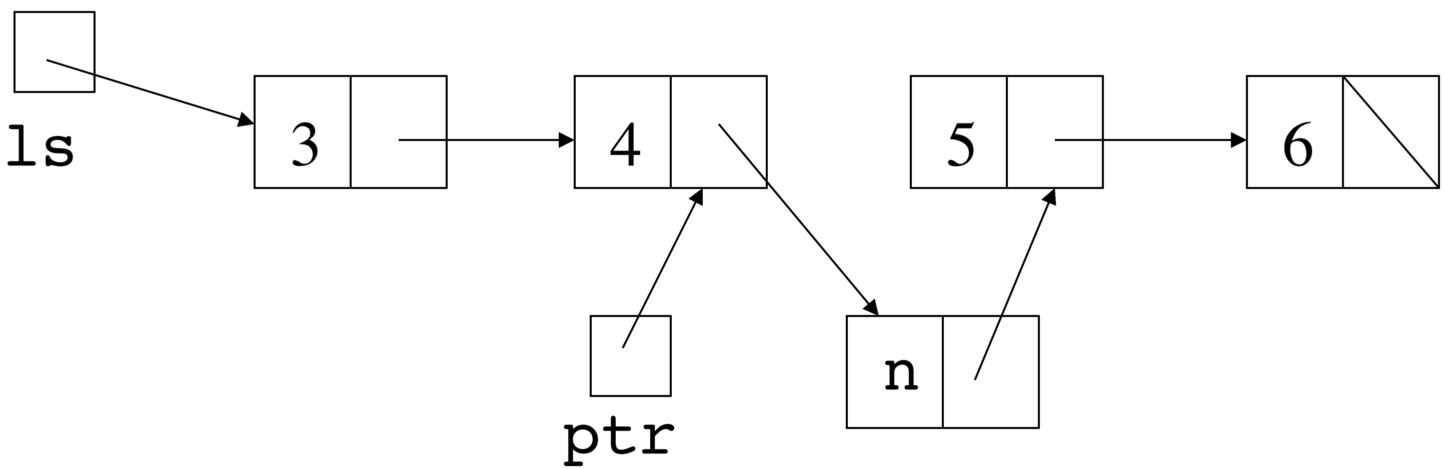
Adding a new cell into a linked list

- If `ptr` points to a cell in a linked list, `ptr.next=new Cell(n,ptr.next)` will add a new cell containing `n` after it



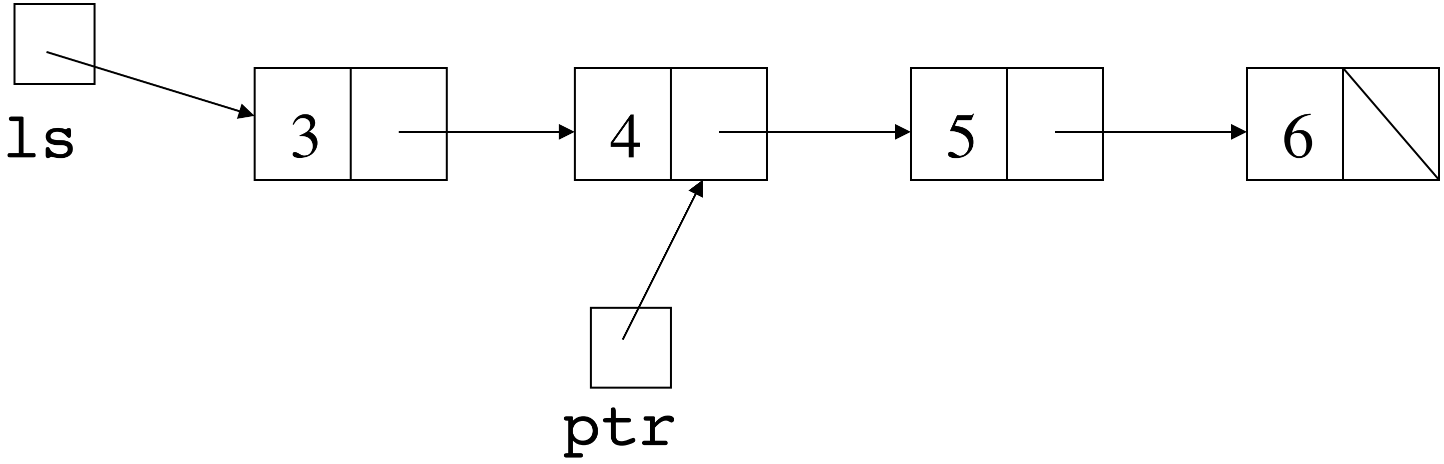
Adding a new cell into a linked list

- If `ptr` points to a cell in a linked list,
  `ptr.next=new Cell(n,ptr.next)`
  will add a new cell containing `n` after it



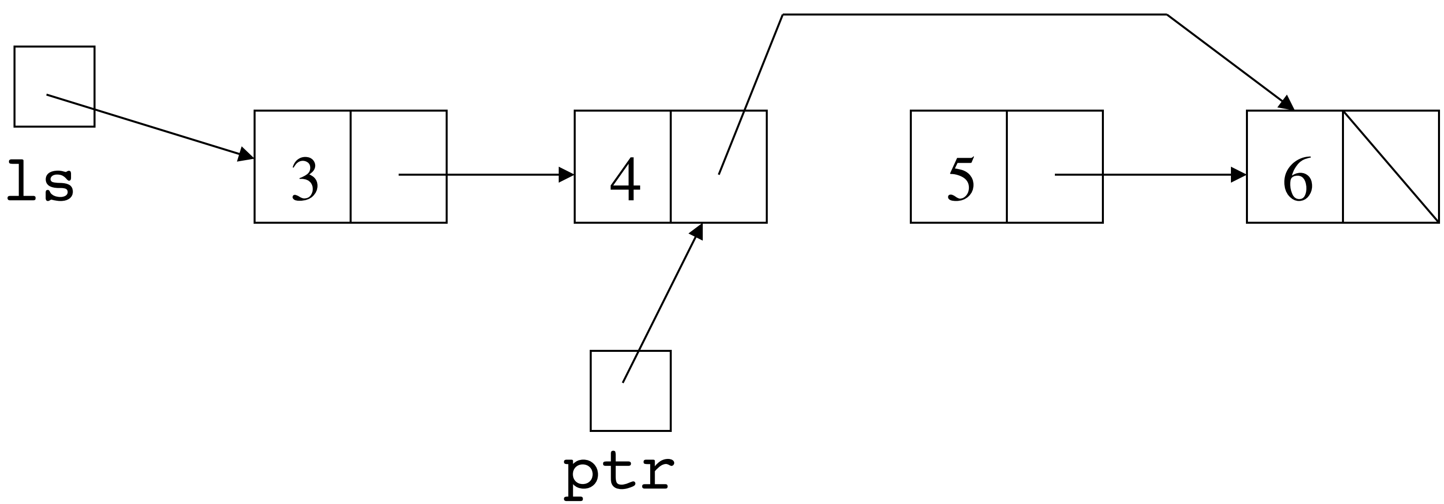
Deleting a cell from a linked list

- If `ptr` points to a cell in a linked list, `ptr.next=ptr.next.next` will delete the following cell from the list



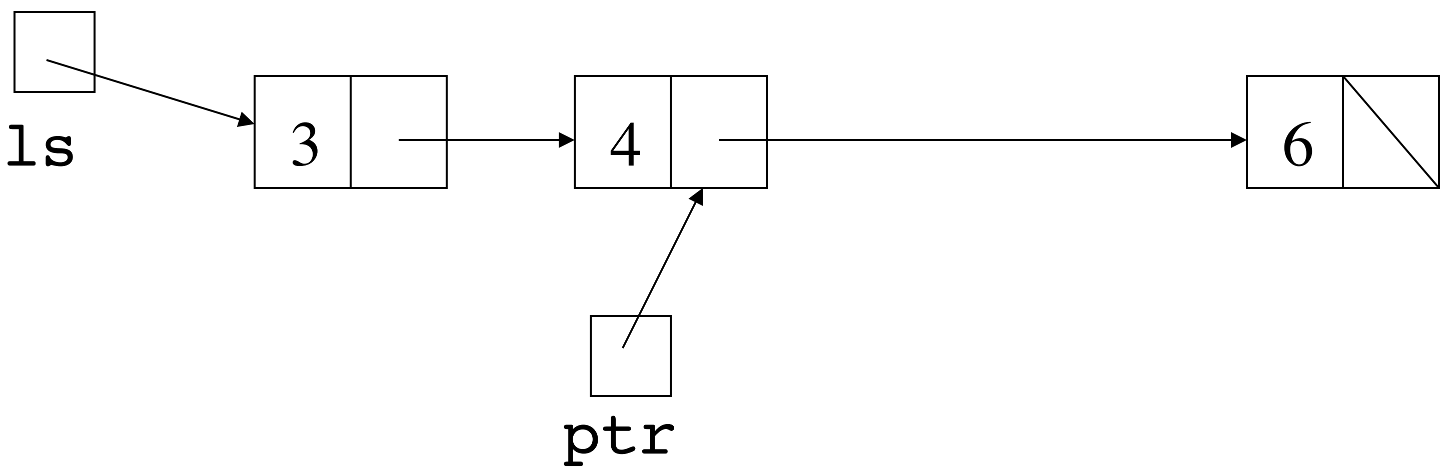
Deleting a cell from a linked list

- If `ptr` points to a cell in a linked list, `ptr.next=ptr.next.next` will delete the following cell from the list



Deleting a cell from a linked list

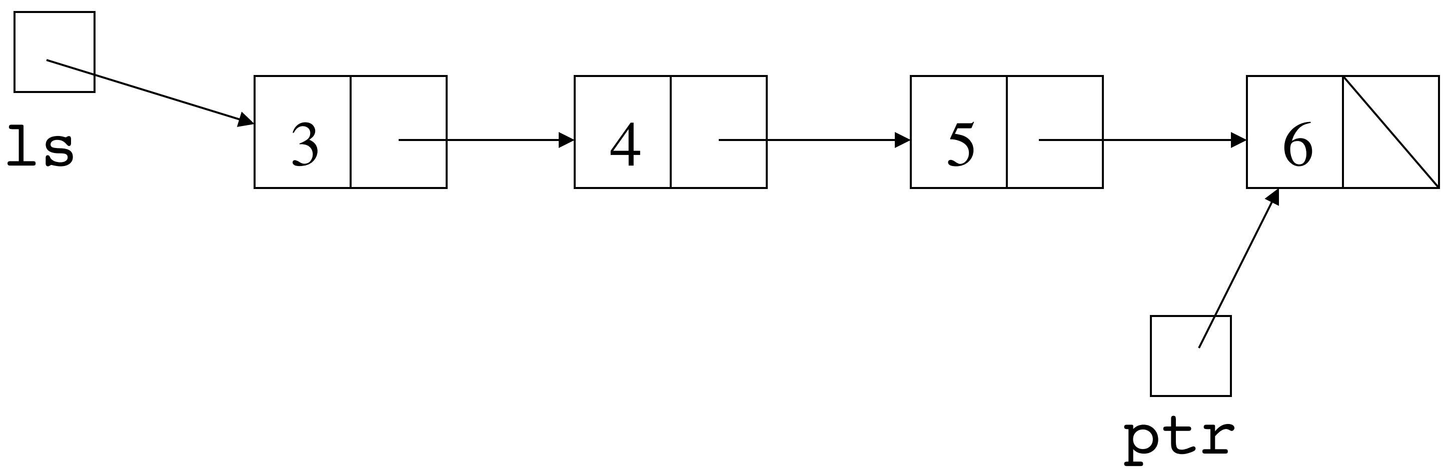
- If `ptr` points to a cell in a linked list, `ptr.next=ptr.next.next` will delete the following cell from the list



Adding an item to the end of a linked list

```
for(ptr=ls; ptr.next!=null; ptr=ptr.next)
  {}
```

will set `ptr` to the last cell



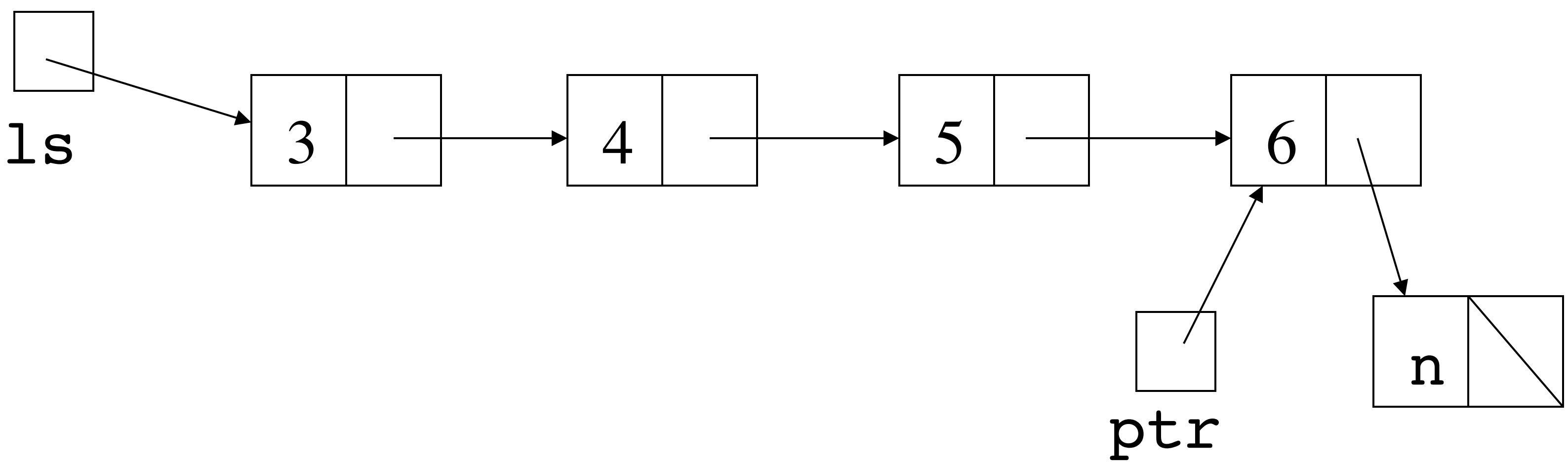
Adding an item to the end of a linked list

```
for(ptr=ls; ptr.next!=null; ptr=ptr.next)
  {}
```

will set `ptr` to the last cell, then

```
ptr.next = new Cell(n,null);
```

adds `n` to the end



Special case for empty list

```
for(ptr=ls; ptr.next!=null; ptr=ptr.next)
  {}
```

What happens if `ls` is `null`?

Special case for empty list

```
for(ptr=ls; ptr.next!=null; ptr=ptr.next)
  {}
```

What happens if `ls` is `null`?

`for` loop does initialisation, then test, if test succeeds then does body and update and repeats

Special case for empty list

```
for(ptr=ls; ptr.next!=null; ptr=ptr.next)
  {}
```

What happens if `ls` is `null`?

`for` loop does initialisation, then test, if test succeeds then does body and update and repeats

So `ptr` becomes `null`, then trying to use `ptr.next` causes a `NullPointerException` to be thrown - not the same as `false` in the test

```
public void add(E item)
{
 if(myList==null)
    myList = new Cell<E>(item,null);
 else
    {
     Cell<E> ptr=myList;
     for(; ptr.next!=null; ptr=ptr.next) {}
     ptr.next = new Cell<E>(item,null);
    }
}
```

Basic code to delete item from linked list

- ```
  for(ptr=ls;
    ptr.next!=null&&!item.equals(ptr.next.first);
    ptr=ptr.next) {}
  ptr.next=ptr.next.next;
  ```
- Needs special case for `ls==null`
- Needs special case for `item.equals(ls.first)`
- Needs special case for loop ends with `ptr.next==null`
  ```

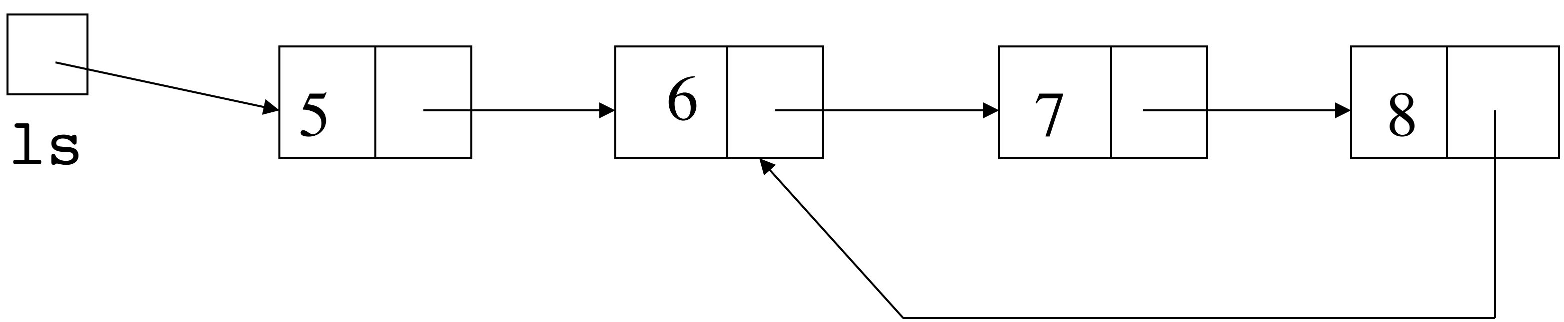
```java
public boolean remove(E item)
{
 if(myList==null)
    return false;
 else if(item.equals(myList.first))
    {
     myList=myList.next;
     return true;
    }
 else
    {
     Cell<E> ptr=myList;
     for(; ptr.next!=null&&!item.equals(ptr.next.first); ptr=ptr.next)
      {}
     if(ptr.next==null)
        return false;
     else
        {
         ptr.next = ptr.next.next;
         return true;
        }
    }
}
```

Programming with linked structures

- Linked structure code can get quite complex
- Code to deal with special cases can dominate, missing it leads to `NullPointerException`s
- Destructive change combined with shared cells could lead to hard to find errors
- So limit use of linked structures to implement carefully designed abstract data types

Circular linked lists

- It is possible to create a linked list in which a link refers to a cell already referred to by another link



- The possibility of circular lists is an additional special case to check for if writing code which directly manipulates linked lists
- But we know none of our code inside our ArrayList and LispList implementations can create a circular linked list, so it's not a problem that code has to deal with
- If we did have that problem, code which deals with linked lists with loops relies on alias checking: `ptr1==ptr2` is `true` if `ptr1` and `ptr2` point to the same cell, `false` otherwise

Code which checks for loop in linked list

```
static boolean containsLoop(Cell<E> ls)
{
int count1=0;
for(Cell<E> ptr1=ls; ptr1!=null; ptr1=ptr1.next,count1++)
  {
   int count2=0;
   for(Cell<E> ptr2=ls; count2<count1; count2++,ptr2=ptr2.next)
       if(ptr2==ptr1)
          return true;
  }
return false;
}
```

More efficient code to check for loop in linked list

```
static boolean containsLoop(Cell<E> ls)
{
 if(ls==null)
   return false;
 for(Cell<E> ptr1=ls, ptr2=ls.next; ptr1!=ptr2; ptr1=ptr1.next)
  {
   if(ptr2==null || ptr2.next==null)
      return false;
   ptr2=ptr2.next.next;
  }
 return true;
}
```

The `equals` method

- The definition of many built-in methods, such as `remove` in `ArrayList`, relies on the `equals` method
- How `equals` works depends on how it is defined for the actual objects being tested
- Default is that a class inherits `Object`'s `equals`
- which means `t1.equals(t2)` gives the same as `t1==t2`

Defining our own `equals`

- We might want `equals` to be more than an alias test
- Consider `LispList<Integer> ls1,ls2`

  we would want `ls1.equals(ls2)` to return `true` if they are separate objects which contain the same integers in the same order
- So in class `LispList` we could write our own `equals` method to override `Object`'s
- Doing this makes sense for immutable objects
- Defining our own `toString` is good as well

`equals` for `LispList<E>`

```
public boolean equals(Object other)
{
 if(!other instanceof LispList)
     return false;
 LispList<E> otherList = (LispList) other;
 if(this.isEmpty())
    return otherList.isEmpty();
 else if(otherList.isEmpty())
    return false;
 else
    return this.head().equals(otherList.head()) &&
           this.tail().equals(otherList.tail());
}
```

`equals` for `LispList<E>` using internal representation

```
public boolean equals(Object other)
{
 if(!(other instanceof LispList))
    return false;
 Cell<E> ptr1 = this.myList;
 Cell<E> ptr2 = ((LispList) other).myList;
 for(;ptr1!=null&&ptr2!=null;
      ptr1=ptr1.rest,ptr2=ptr2.rest)
    {
     if(!ptr1.first.equals(ptr2.first))
       return false;
    }
 return (ptr1==null&&ptr2==null);
}
```

`equals` for `LispList<E>` using shared cells

```
public boolean equals(Object other)
{
 if(!(other instanceof LispList))
    return false;
 Cell<E> ptr1 = this.myList;
 Cell<E> ptr2 = ((LispList) other).myList;
 for(;ptr1!=ptr2&&ptr1!=null&&ptr2!=null;
      ptr1=ptr1.rest,ptr2=ptr2.rest)
    if(!ptr1.first.equals(ptr2.first))
       return false;
 return (ptr1==ptr2);
}
```

ArrayList implementation using linked list with size variable

```
class MyArrayList<E>
{
 private Cell<E> myList;
 private int mySize;

 private MyArrayList()
 {
  myList=null;
  mySize=0;
 }

...
```

Why?

- We can work out the size of an ArrayList represented by a linked list by sending the pointer down the linked list and counting the number of times we do a `ptr=ptr.next` until `ptr` becomes `null`
- This is inefficient
- A separate size variable, updated in any method which changes the size of the ArrayList, is redundant in terms of necessity, but valuable in terms of efficiency

- Used to implement `size()`

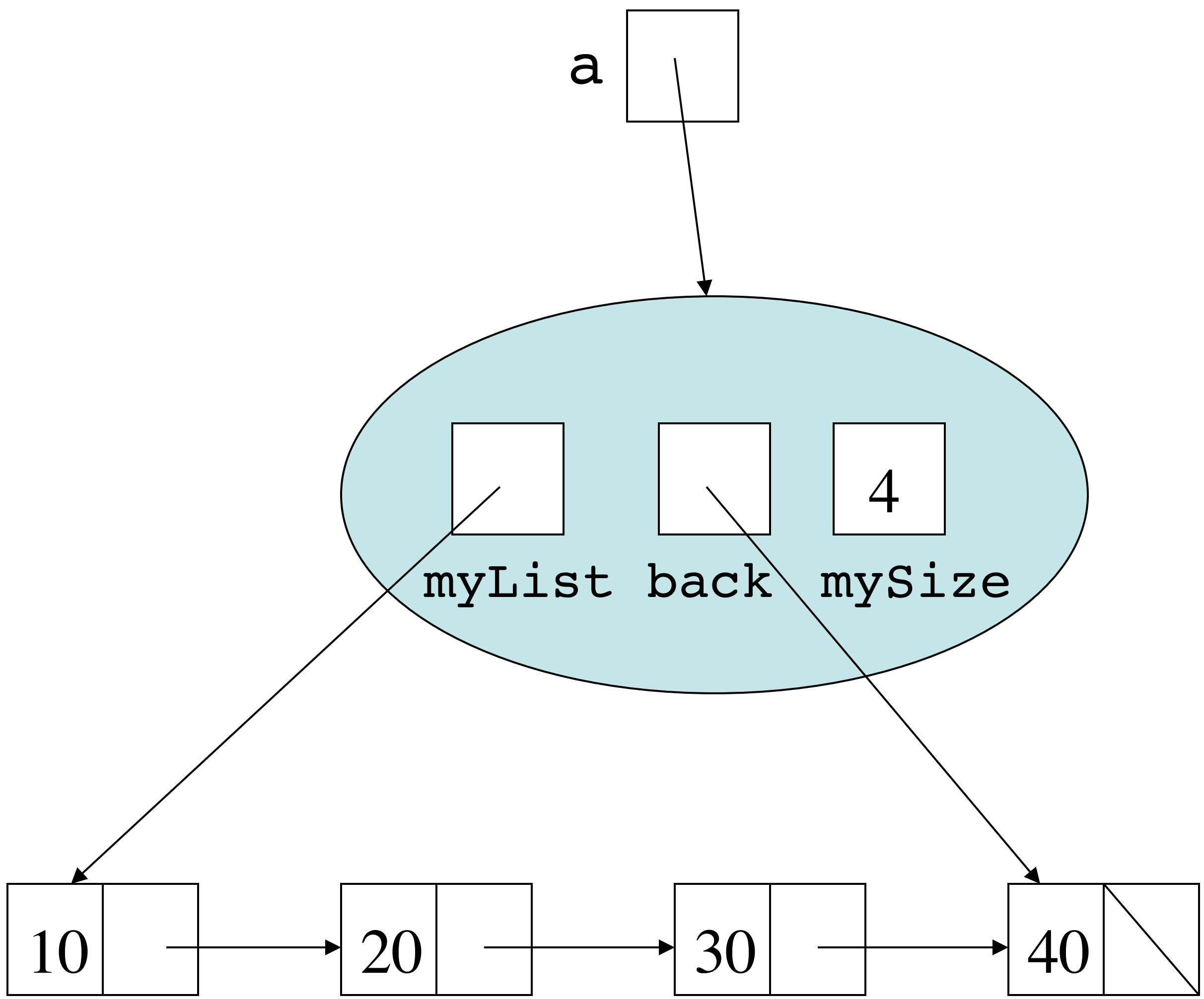
```
public int size()
{
 return mySize;
}
```

- Used to prevent unnecessary list traversal when an `IndexOutOfBoundsException` should be thrown

```
public E get(int pos)
{
 Cell<E> ptr=myList;
 if(pos>=mySize)
       throw new IndexOutOfBoundsException();
 for(int count=0; count<pos; ptr=ptr.next, count++) {}
 return ptr.first;
}
```

ArrayList implementation with back pointer

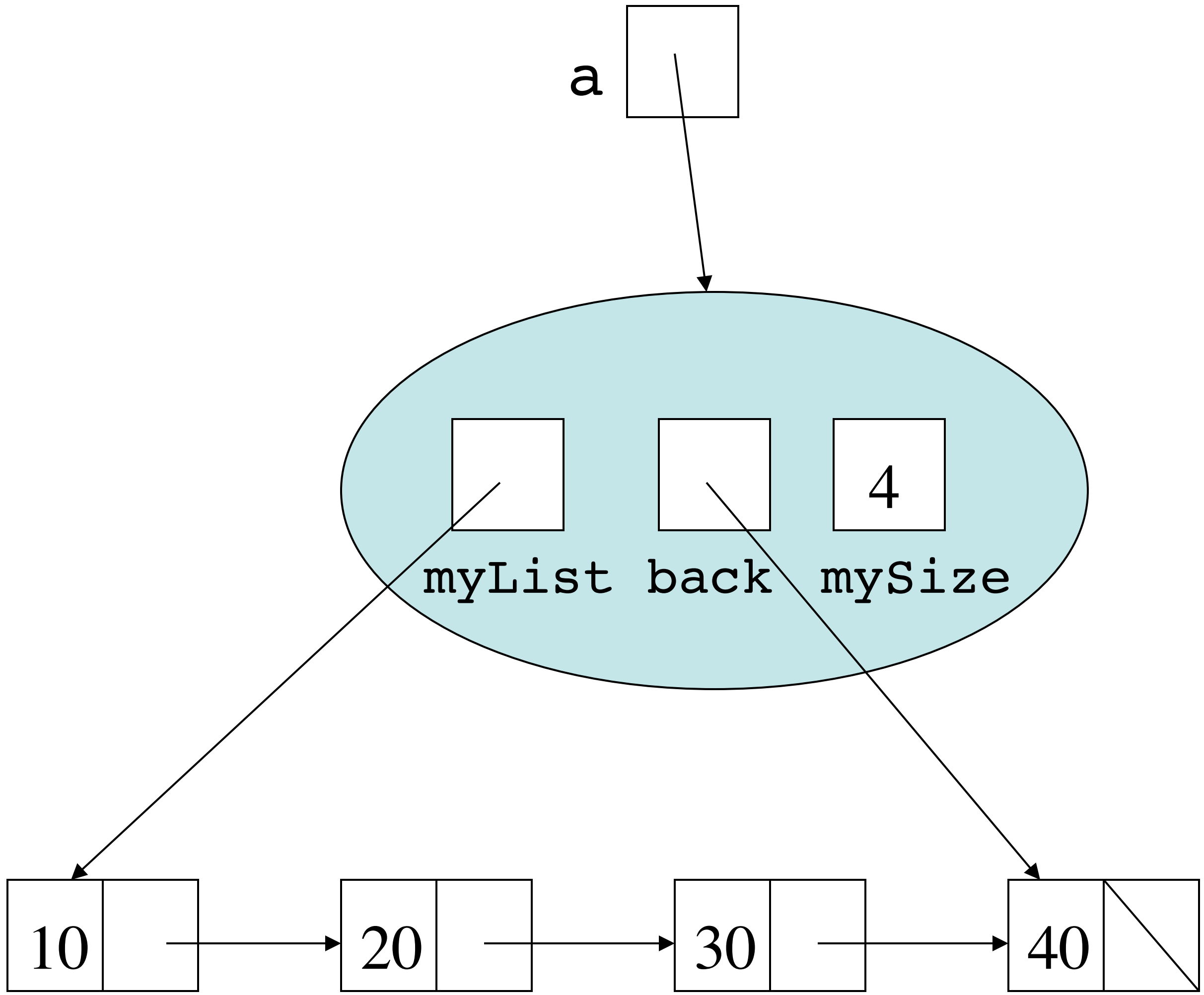
```
class MyArrayList<E>
{
 private Cell<E> myList,back;
 private int mySize;

 private MyArrayList()
 {
  myList=null;
  back=null;
  mySize=0;
 }

...
```

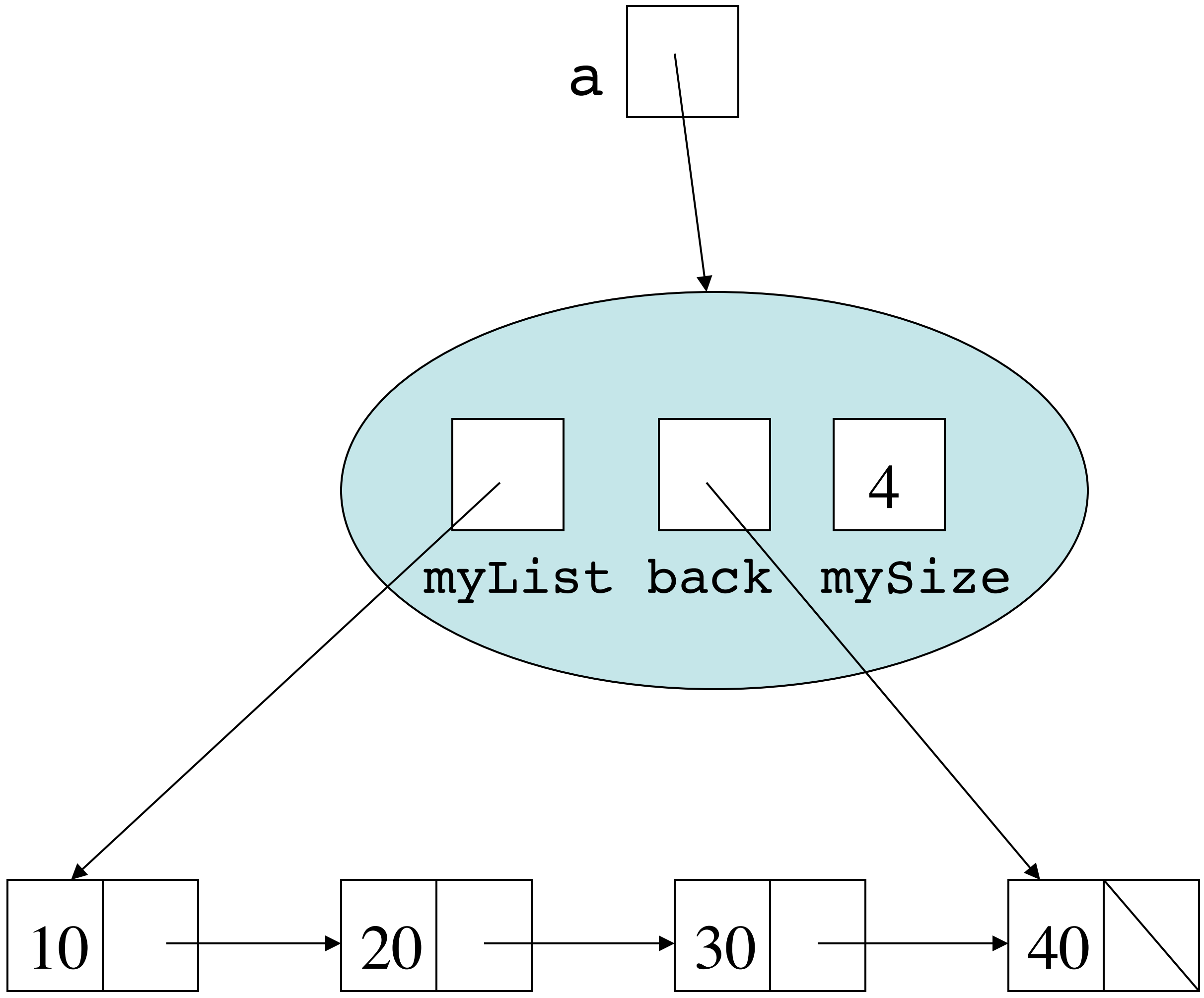

a
4
myList
back
mySize
10
20
30
40

Why?

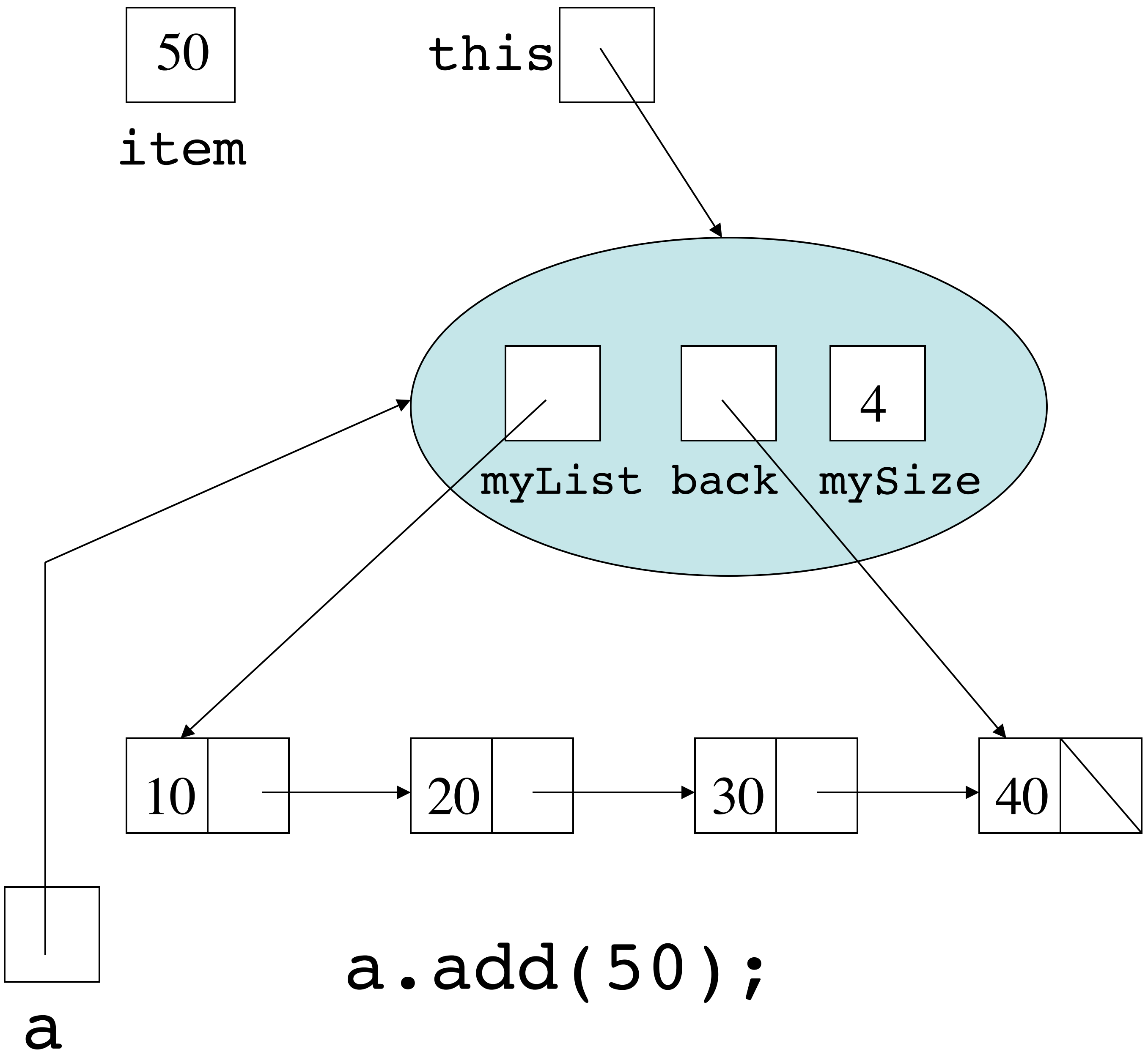
- The method `add(n)` is commonly called on ArrayList objects
- With a pointer to the back cell, we do not have to traverse all the cells to add a new item to the end
- This means that `a.add(n)` takes the same amount of time, no matter what the size of the ArrayList referred to by `a`

```
public void add(E item)
{
 if(myList==null)
     {
      myList = new Cell<E>(item,null);
      back = myList;
     }
 else
    {
     back.next = new Cell<E>(item,null);
     back = back.next;
    }
 mySize=mySize+1;
}
```

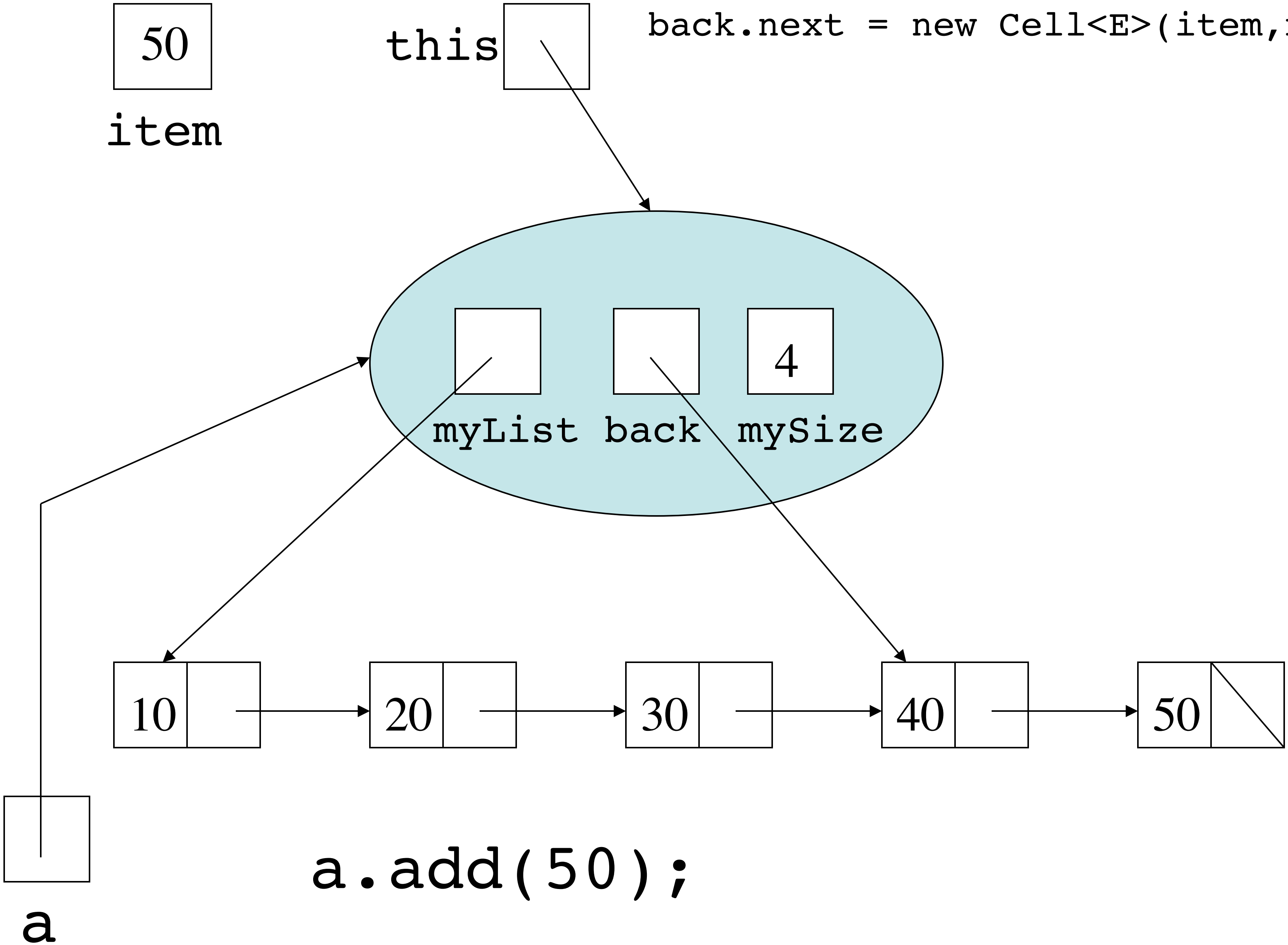

a
4
myList back mySize
10
20
30
40




```
a.add(50);
```

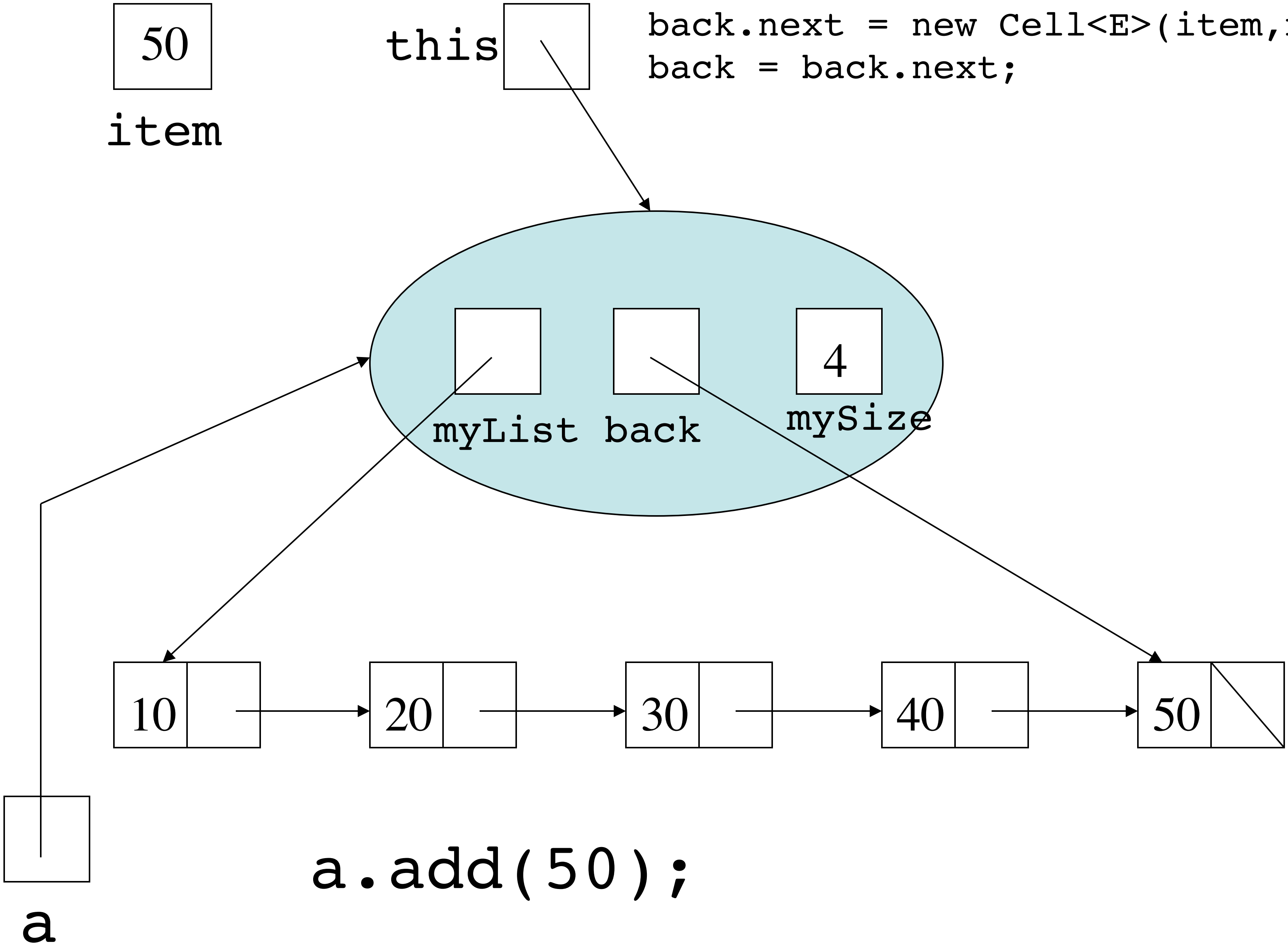

50
item
this
myList
back
4
mySize
10
20
30
40
a
a.add(50);

```java
back.next = new Cell<E>(item,null);
```



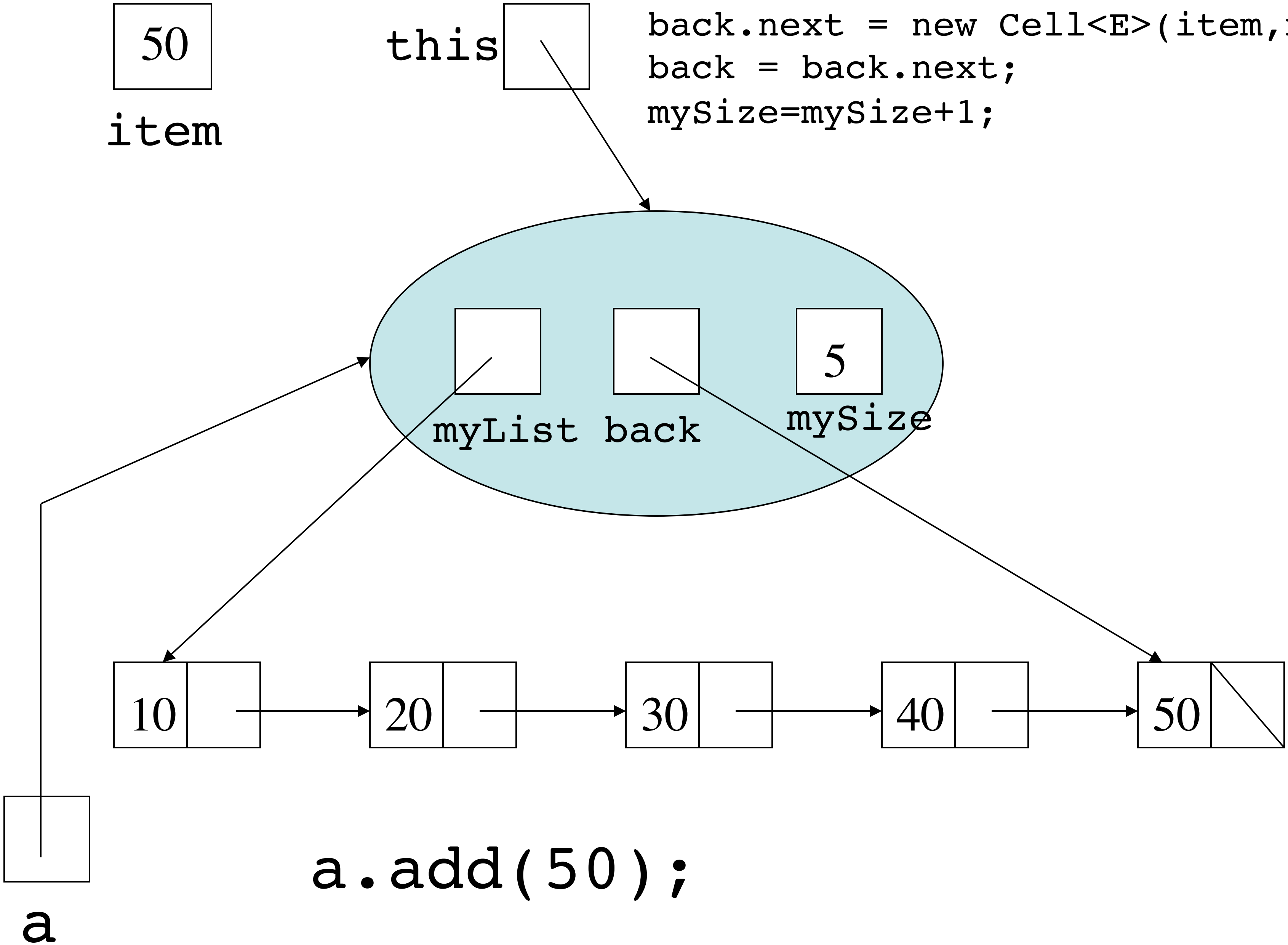

```java
a.add(50);
```

```
back.next = new Cell<E>(item,null);
back = back.next;
```

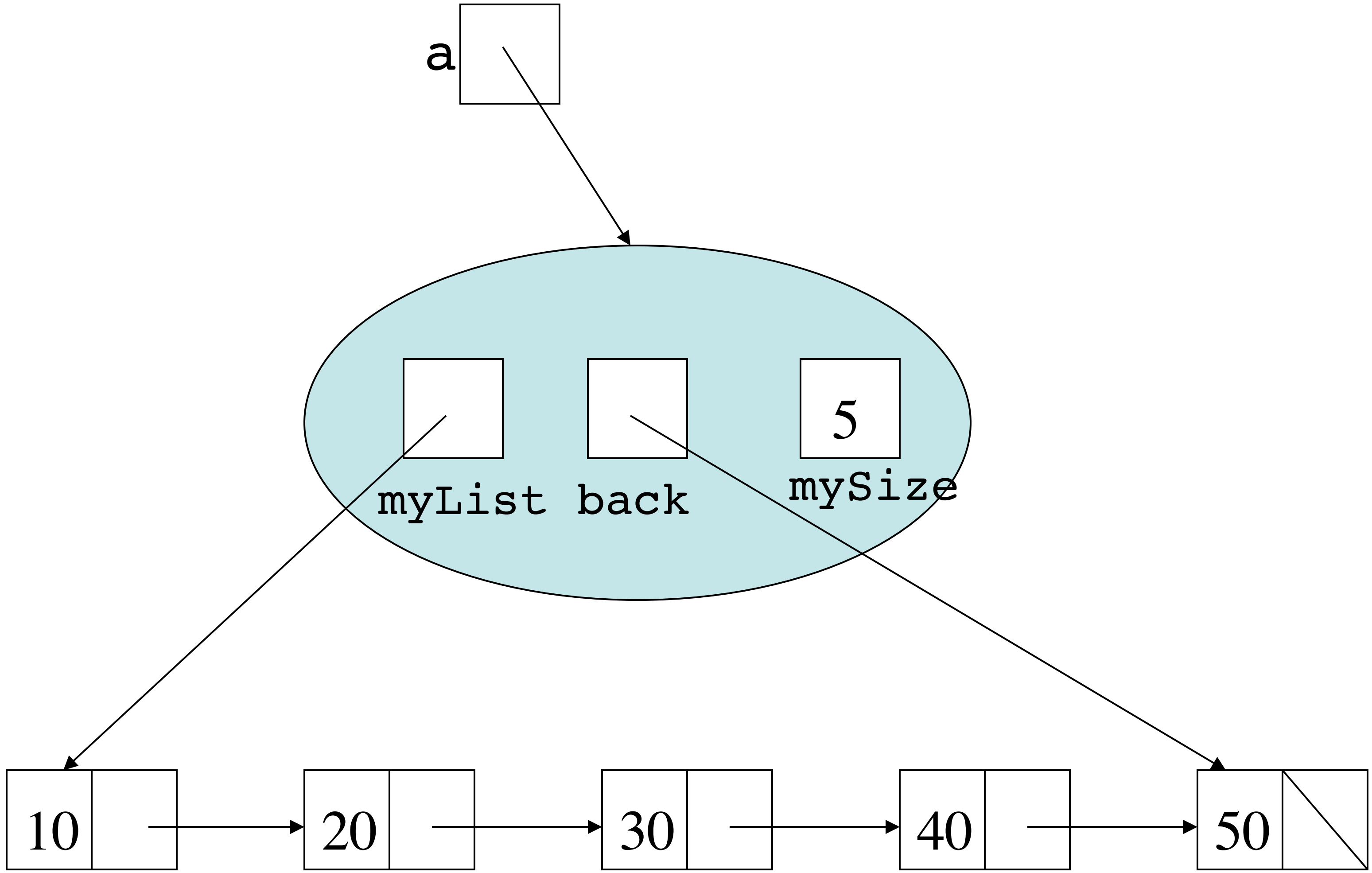



```
a.add(50);
```

```
back.next = new Cell<E>(item,null);
back = back.next;
mySize=mySize+1;
```




```
a.add(50);
```


a
5
myList
back
mySize
10
20
30
40
50

Doubly Linked lists

- A linked list structure where each cell has <u>two</u> cell variables
- The two cell variables, called `next` and `prev`, have to be set so that the following always holds after each method call for `ptr` pointing to any cell:

  `ptr.next.prev==ptr` unless `ptr.next==null`

  `ptr.prev.next==ptr` unless `ptr.prev==null`
- The result is a list in which the `next` links point forward and the `prev` links point backwards

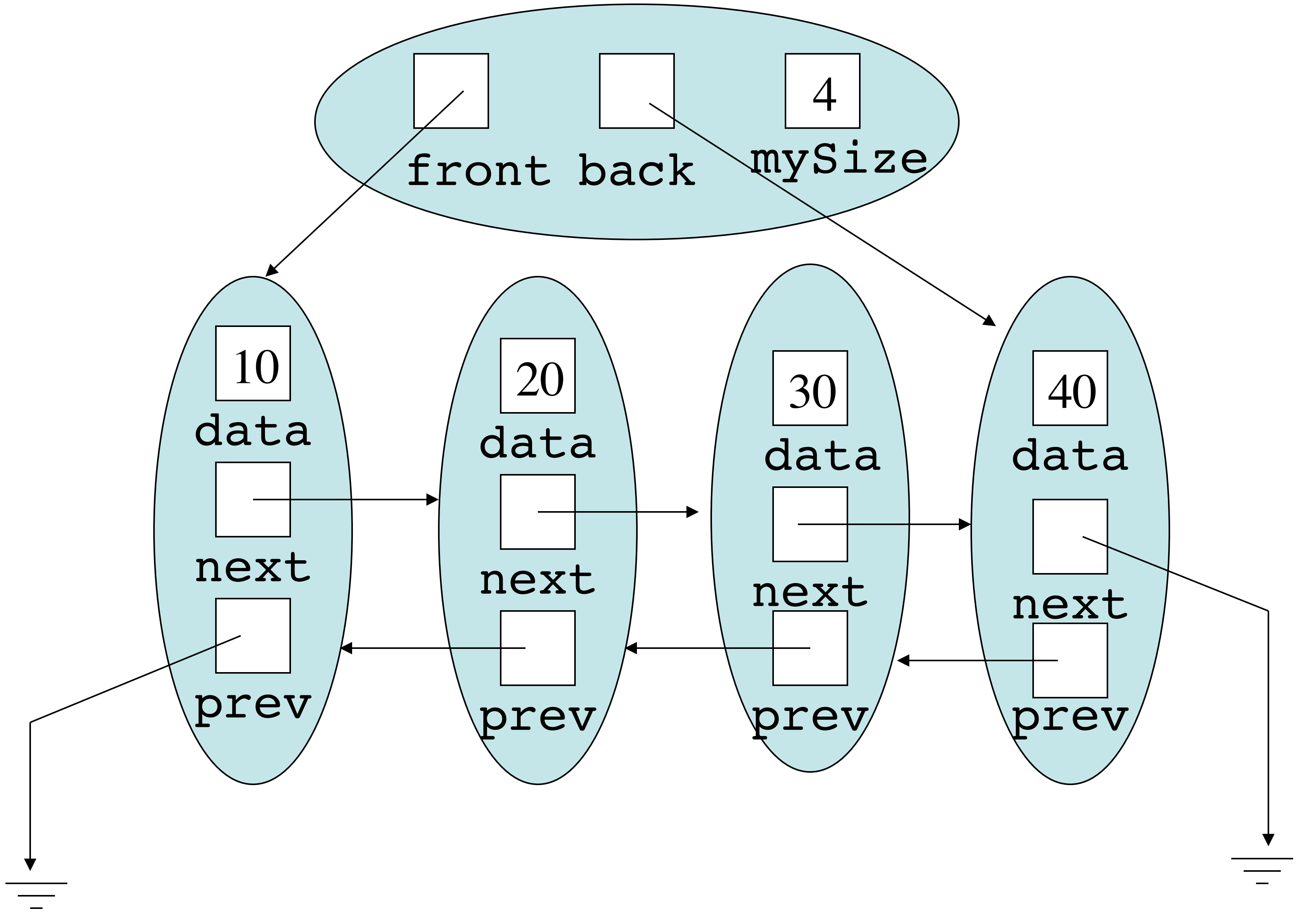
```
private static class DCell <T>
{
 T data;
 DCell<T> next,prev;

 DCell(T d,DCell<T> n,DCell<T> p)
 {
  data=d;
  next=n;
  prev=p;
 }
}
```

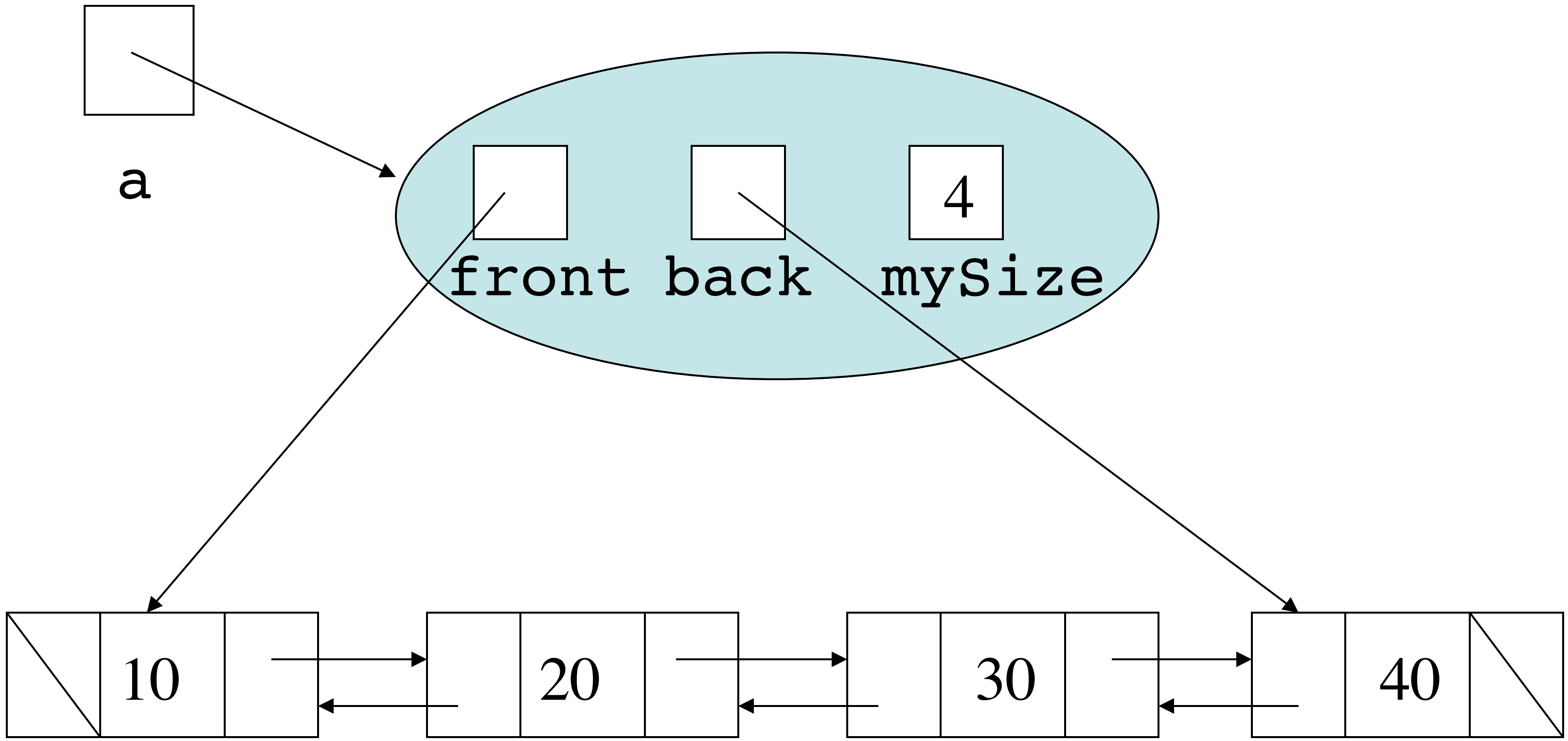
Binary trees

- A structure consisting of cells each of which has two variables linking to further cells is called a “binary tree” if the following holds:
  - One cell (the “root”) has no other cells pointing to it
  - All other cells have exactly one cell pointing to them
  - So as cell variables may be set to `null`, each cell stores a data item and links to 0, 1 or 2 further cells
- This is an important data structure, but we do not have time to cover it in this module

Implementing ArrayList with doubly-linked list

```
class MyArrayList <E>
{
 private DCell<E> front,back;
 private int mySize;


  …
```


4
front
back
mySize
10
data
next
prev
20
data
next
prev
30
data
next
prev
40
data
next
prev


a
4
front
back
mySize
10
20
30
40

```
public void add(int pos,E item) {
 if(pos>mySize)
    throw new IndexOutOfBoundsException();
 if(pos==0) {
     front = new DCell<E>(item,front,null);
     if(mySize==0)
        back=front;
     else
        front.next.prev=front;
    }
 else {
     DCell<E> ptr=front;
     for(int count=1; count<pos; ptr=ptr.next,count++) {}
     ptr.next = new Cell<E>(item,ptr.next,ptr);
     if(ptr==back)
        back=ptr.next;
     else
        ptr.next.next.prev=ptr.next;
    }
 mySize=mySize+1;
}
```

Why?

- Efficient for adding and removing new items at either end of the list
- For items inside, we can start at the closest end
- But this is not as efficient access as an array implementation (one step for any position)
- However, it avoids the “moving up/down” when items are added/removed from inside the list

Java's `List` and `LinkedList`

- Java has an interface type `List<E>`
- `ArrayList<E>` implements `List<E>` using array and count
- `LinkedList<E>` implements `List<E>` using doubly-linked list
- Rather than use `ArrayList<E>`, use `List<E>`
- Use `ArrayList<E>` or `LinkedList<E>` only when constructing new objects

Why?

- A `List<E>` variable can refer to an `ArrayList<E>` object or a `LinkedList<E>` object, so your code is generalised
- When creating a new `List<E>` object, you can use whichever implementation will be more efficient for the purposes you want to use it for
- `LinkedList<E>` provides extra methods for accessing the ends of a list
- `ArrayList<E>` provides extra methods for fine-tuning the array underneath

Implementation and Application

- A clear distinction between implementation and application hides complex data structure code
- It ensures data structures are not manipulated into unacceptable formats by outside code
- It enables us to pick and choose between different implementations for efficiency reasons
- It enables us to write generalised code which is not dependent on any particular implementation