Algorithms and Data Structures (ECS529)

Nikos Tzevelekos

Lecture 8

Linked Lists

Remember lists

Another standard data structure is that of a **list**:

- the elements of a list are stored in a sequence
- we can read/write the elements of a list (not necessarily via indexes)
- we can add new elements at any position inside a list, or remove existing ones, changing the length of the list

We previously saw array lists, which are an extension of arrays that work as lists.

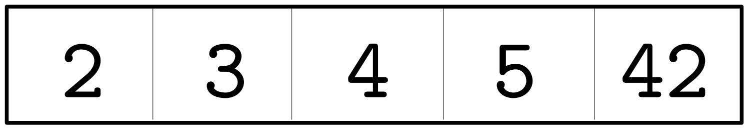
In this lecture we will study **linked lists**.

These introduce the notion of a **linked data structure**: a data structure that uses pointers to hold its data together.

Linked lists

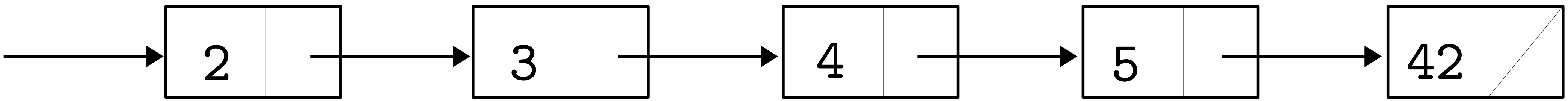
Data structures = ways to give structure to a ‘soup’ of data.

- Arrays do this by putting the data in a line-up of containers:



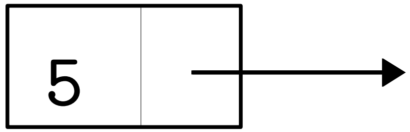
- Linked lists do this by putting the data into a chain of **nodes** where each node stores two things:
  - the data we want to store
  - a pointer (or arrow) to the next node in the chain

We typically draw linked lists like this:



Pointers

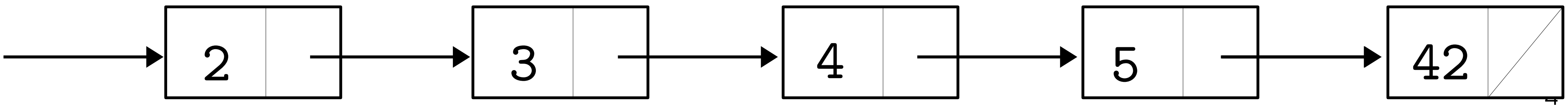
What is a pointer to another node?



- it is simply the computer **address** of the next node
- said otherwise, it is a **reference** to the next node

If there is no next node, we simply store some ‘null’ address (`None` in Python, `null` in Java, etc.). So, null pointers determine the **end** of the list (e.g. 42 below).

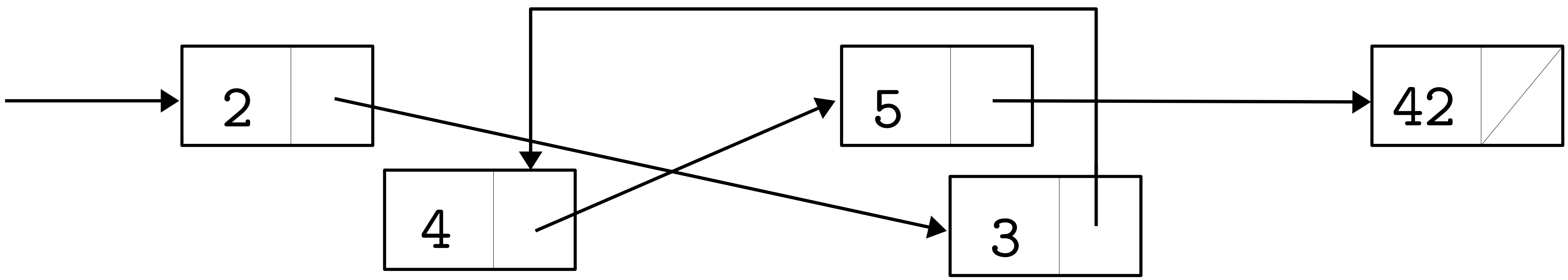
The beginning, or **head**, of the list is determined by an incoming arrow with no source node.



Are these not just arrays?

They are not!

In linked lists there is no reason why nodes should be stored in sequence. E.g. this is perfectly legit:



So, linked lists are like soups with pointers. We will see that:

- pointers make it easy to add/remove elements
- but make more complicated reaching an element in a list

Nodes of linked lists

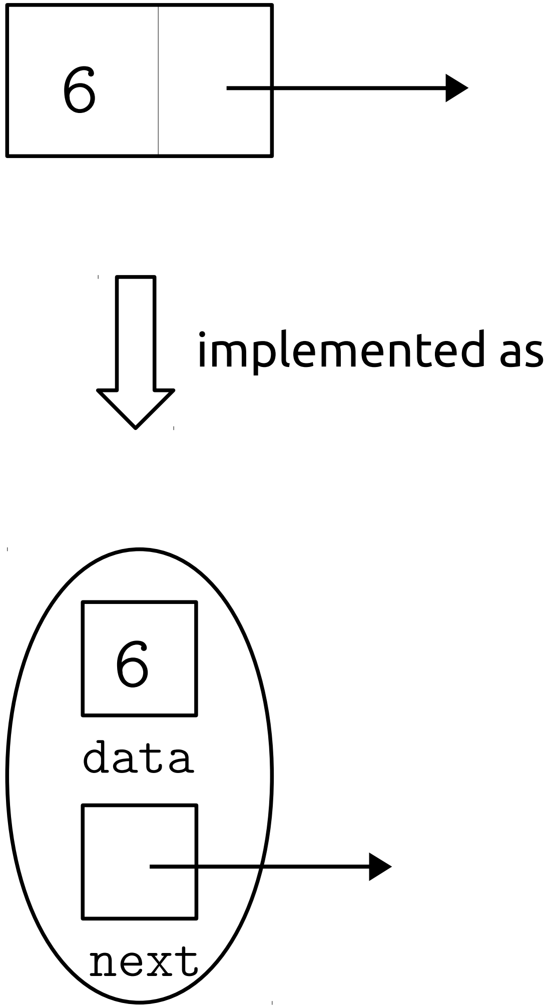
Here is an implementation of nodes in Python:

```
class Node:
  def __init__(self, d, n):
    self.data = d
    self.next = n
```

and here it is in Java:

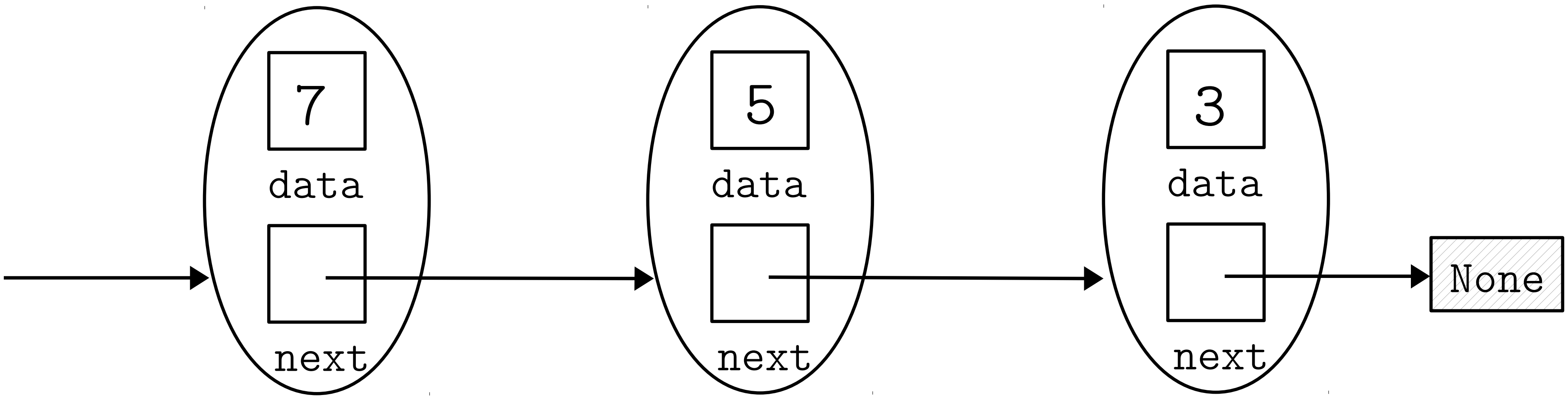
```
class Node<T> {
  T data;
  Node<T> next;

  Node(T d, Node<T> n) {
    data = d;
    next = n;
  }
}
```

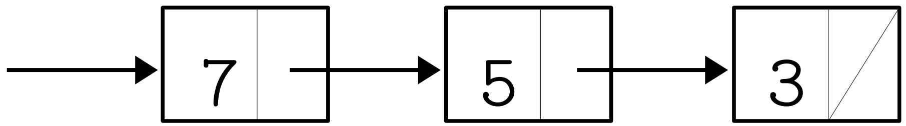


Example linked list

Here is how the list [7, 5, 3] is implemented:



which we draw simply as:



Linked list creation

We create a list by:

- starting from the empty list (`None`)
- and adding elements to it.

E.g. the list [7, 5, 3] is created by:

```
ls = Node(3,None)
ls = Node(5,ls)
ls = Node(7,ls)
```

So, in this case we keep adding nodes to the **head** of the list, i.e. its beginning.

Note:
when writing, for example

```
ls = Node(5,ls)
```

we mean:

- create a new node with data 5 and with next pointing to the node stored in `ls`
- store the new node in `ls`

i.e. the same as:

```
tmp = Node(5,ls)
ls = tmp
```

Linked list creation

We can also create a list starting from empty and adding elements at the end of it. E.g. the list [7, 5, 3] is created by:

```
ls = Node(7,None)
ls.next = Node(3,None)
ls.next.next = Node(3,None)
```

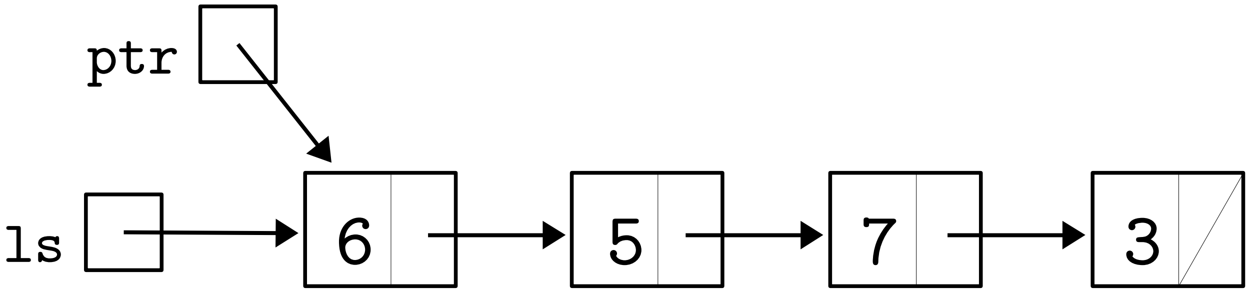
This method is not great for long lists:

- we want to add things to the end of the list but `ls` points to the first node in the list!
- we can try stacking up `nexts` but that is not scalable – we will next see methods for traversing the list to reach its end

Moving pointers down a list

The point of access of a linked list is the "head" node (leftmost one).

For example, suppose `ptr` points to the head of this list:



To access other nodes, we can move the pointer down the list by:

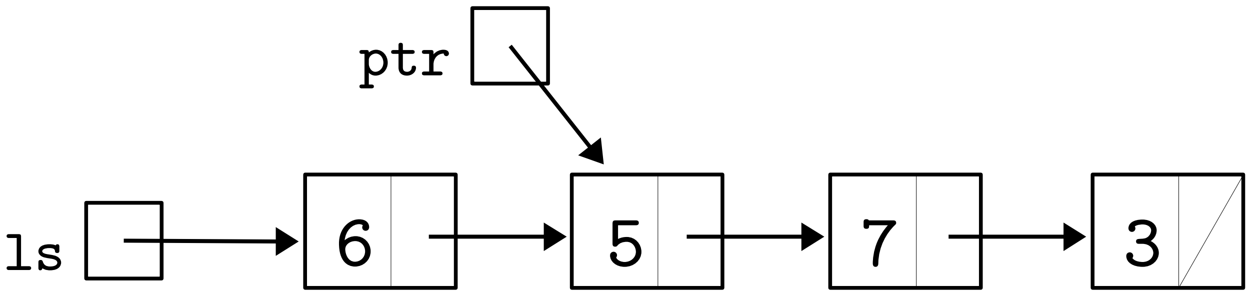
```
ptr = ptr.next
```

this is the linked-list equivalent of

`i = i+1`

that we use in arrays

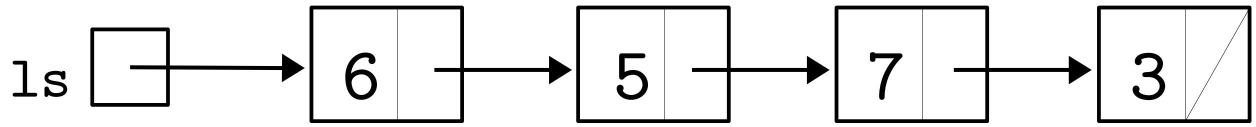
E.g. after performing `ptr = ptr.next` once:



This moving down of pointers is also called *traversing* the linked list.

Example: finding last element in a list

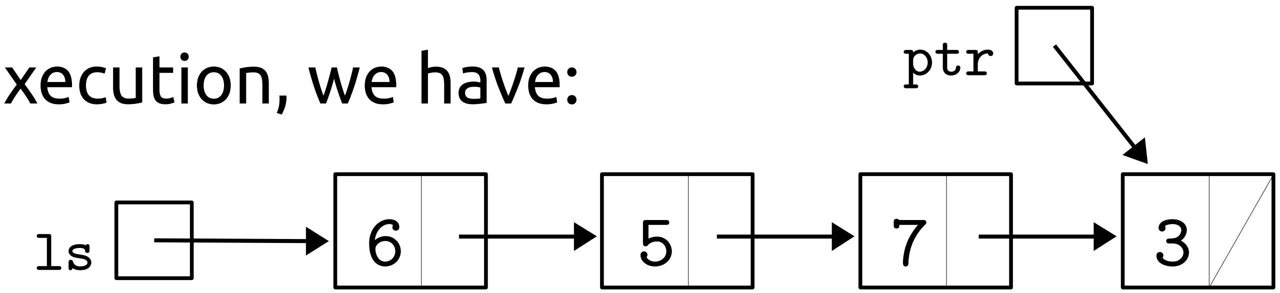
Suppose `ls` is this list:



We can reach the end of the list with this code:

```
ptr = ls
while ptr.next != None:
    ptr = ptr.next
```

after its execution, we have:



Appending elements at (the end of) a linked list

We let the empty linked list (no elements) be simply `None`.

So, we can add an element `d` at the end of the linked list `ls` by:

```
# append d at end of ls
if ls == None:
   ls = Node(d,None)
else:
   ptr = ls
   while ptr.next != None:
      ptr = ptr.next
   ptr.next = Node(d,None)
```

If `ls` is empty then we simply assign `[d]` to it.

Otherwise, we move the pointer `ptr` down `ls` until it points to the last node of `ls`.

We insert `d` by:

- creating a new node `(d,None)`
- setting `ptr.next` to that new node

Searching for an element in a linked list

We can use the same technique to write code that returns the index of the first node inside `ls` where a specific element `d` appears (note: the first element of `ls` has index 0, the second has 1, and etc.).

If `d` does not appear in `ls` then we return -1.

```
def search(d):
    i = 0
    ptr = ls
    while ptr != None:
        if ptr.data == d:
            return i
        ptr = ptr.next
        i += 1
    return -1
```

We first set `i` to 0. We then move a pointer `ptr` down `ls` until it points to a node whose data is equal to `d`.

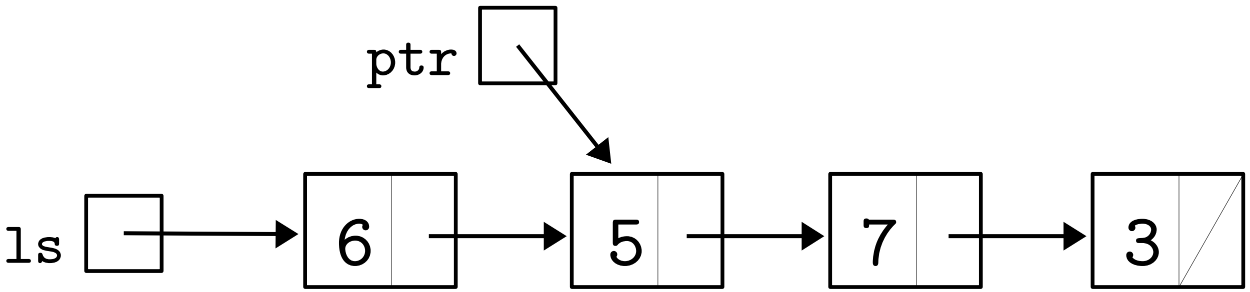
At each move of `ptr` we increase `i` by 1. So, `i` is keeping track of the position of `ptr` in the list.
When we find `d`, we return `i`.

If `ls` is None or if `d` is not found in it then the while loop will stop when `ptr` is None, and we will return -1.

Inserting elements in a linked list

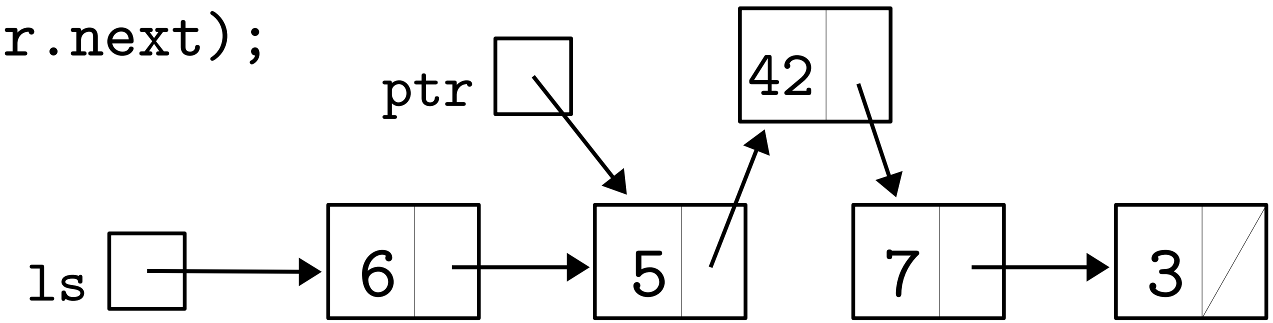
We can add an element at an arbitrary position in a linked list.
For that, we need to have a **pointer to the node immediately before** the point of insertion.

Suppose `ptr` points to an element in a linked list:



To add a new element with value 42 right after `ptr`, we create a new node that stores 42 and intersects the pointers between 5 and 7:

```
ptr.next = Node(42,ptr.next);
```



Inserting element at position i of a linked list

We use the previous technique to add an element `d` at position `i` of the linked list `ls` (0 is the first element of the list, etc.):

```
# inserts d at position i of ls
if ls == None:
   ls = Node(d,None)
elif i == 0:
   ls = Node(d,ls)
else:
   ptr = ls
   while i>1 and ptr.next != None:
      ptr = ptr.next
      i -= 1
   ptr.next = Node(d,ptr.next)
```

If `ls` is empty then we simply assign `[d]` to it.

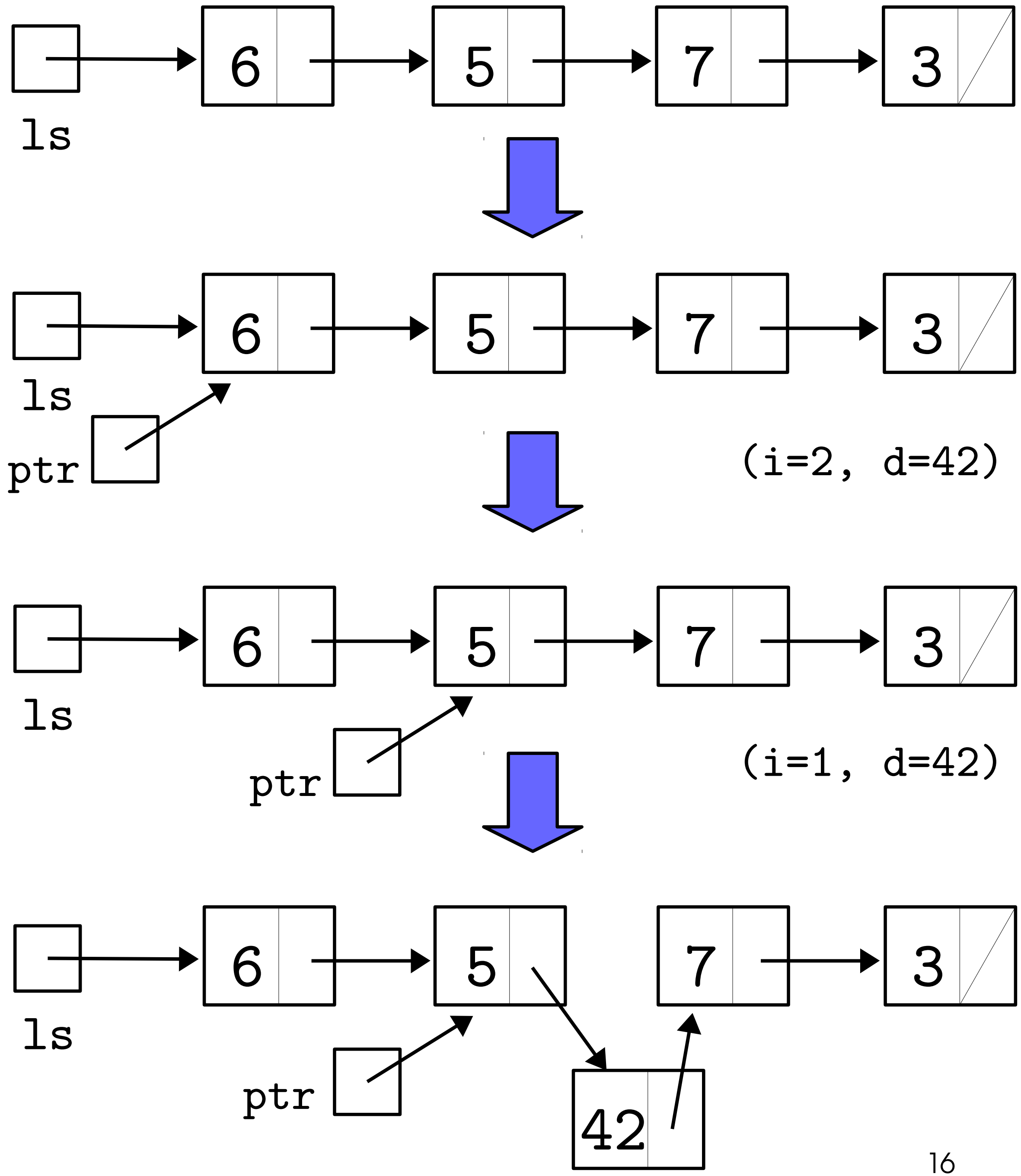
If `i` is 0 then we just need to add `d` at the beginning of `ls`.

Otherwise, we need to move a pointer `ptr` down in `ls` until we find the position **after which** we want to insert `d` (this is when `i` will become 1).

We insert `d` in that position.

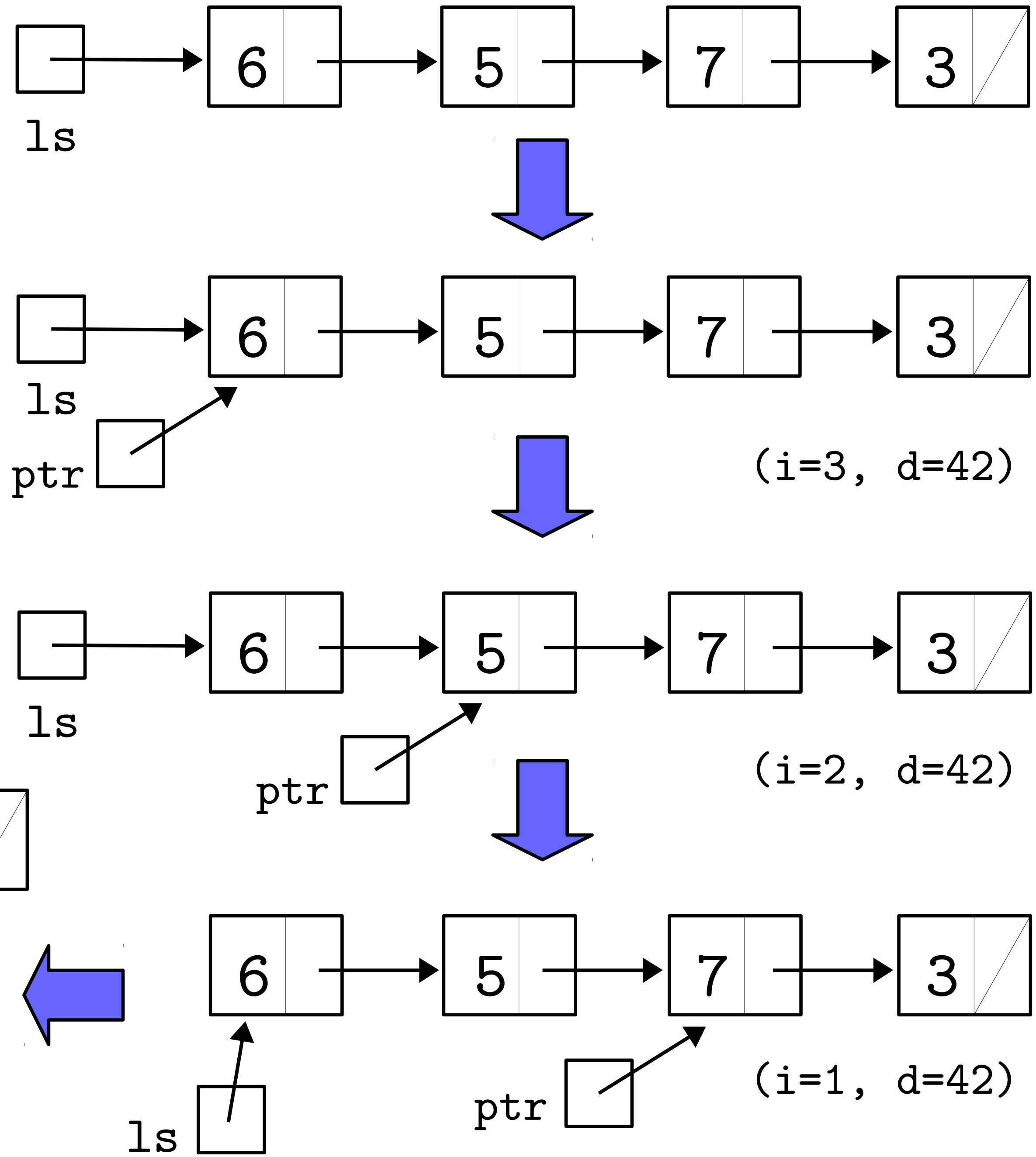
Example of list insertion

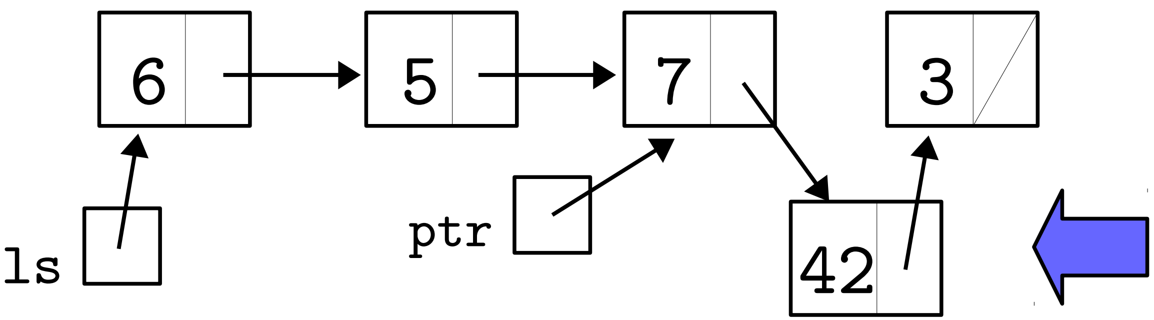
```
i = 2; d = 42
if ls == None:
    ls = Node(d,None)
elif i == 0:
    ls = Node(d,ls)
else:
    ptr = ls
    while i>1 and ptr.next != None:
        ptr = ptr.next
        i -= 1
    ptr.next = Node(d,ptr.next)
```



Example of list insertion

```
i = 2; d = 42
if ls == None:
    ls = Node(d,None)
elif i == 0:
    ls = Node(d,ls)
else:
    ptr = ls
    while i>1 and ptr.next != None:
        ptr = ptr.next
        i -= 1
    ptr.next = Node(d,ptr.next)
```



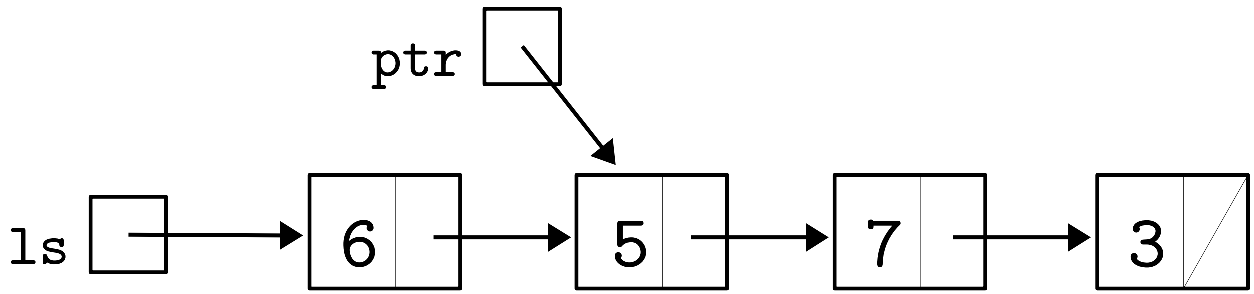


Removing elements from a linked list

Removing of elements amounts to bypassing them!

As was the case with insertion, we need a **pointer to the node before** the one to remove.

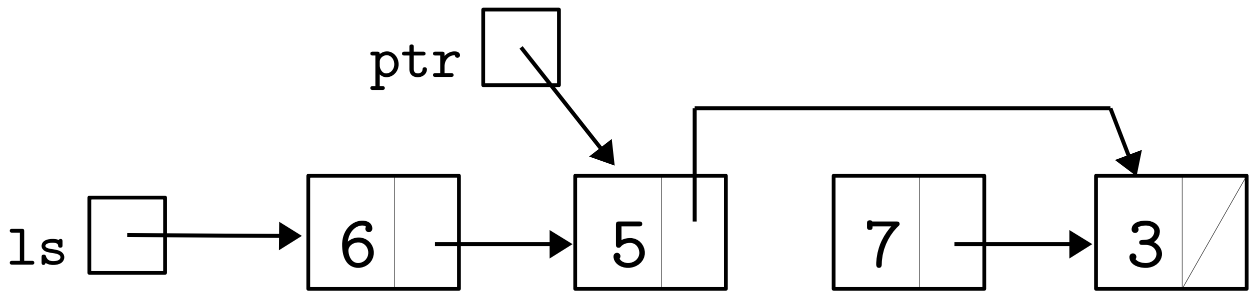
Suppose we have the list `ls` and want to remove the node 7:



we can do this by:

```
ptr.next = ptr.next.next;
```

and we get:



the node is still there (somewhere) but it is not in our list anymore

Removing element from a linked list (1st attempt)

We can use the previous technique to search and remove the first occurrence of an element `d` from a linked list `ls`:

```
# remove the first d found in ls
ptr = search(d)                  # search for d and
if ptr != None:                  # if found then remove it
    ptr.next = ptr.next.next
```

what is wrong with this code? Well:

- searching for `d` will give us a pointer at the (first) node containing it
- but what we need is a pointer to the node just before that

So, we need a different kind of search, namely one that returns the node before the first occurrence of `d`.

Removing an element from a linked list

We can use the previous technique to search and remove the first occurrence of an element `d` from a linked list `ls`:

```
# removes the first d found in ls
if ls == None:
   pass
elif ls.data == d:
   ls = ls.next
else:
   ptr = ls
   while ptr.next != None:
      if ptr.next.data == d:
         ptr.next = ptr.next.next
         break
      ptr = ptr.next
```

If `ls` is empty then we do nothing

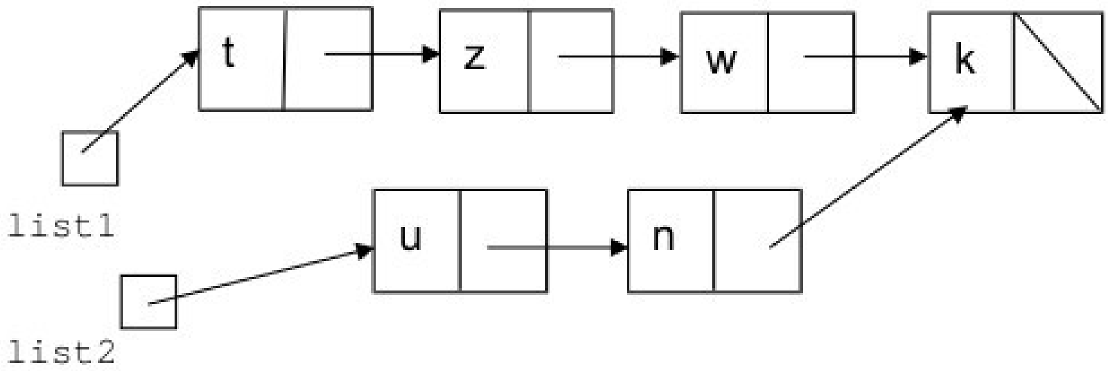
If `d` is in the first node then we just bypass the first node of `ls`

Otherwise, we move a pointer `ptr` down `ls` until we find the position **after which** we have the node that we want to remove (i.e. when we have that `ptr.next.data == d`).

We remove the next node there by bypassing it

Pointer games

Suppose we have created the following lists:



Perform the following operations in sequence and draw the new lists:

a) `list1.next = list1.next.next`

b) `list2.next = list1.next`

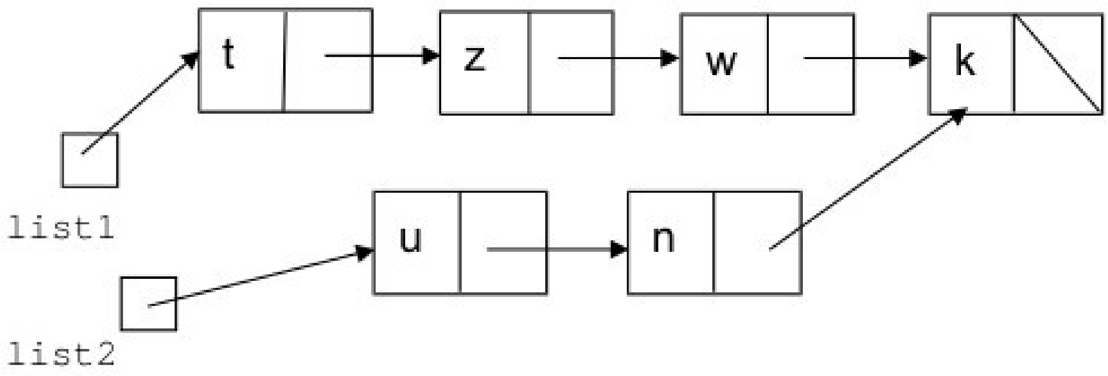
   `list1.next.data = 'p'`

c) `list2.next = new Node('d',list1)`

d) `list1.next = list2`

Pointer games

Suppose we have created the following lists:



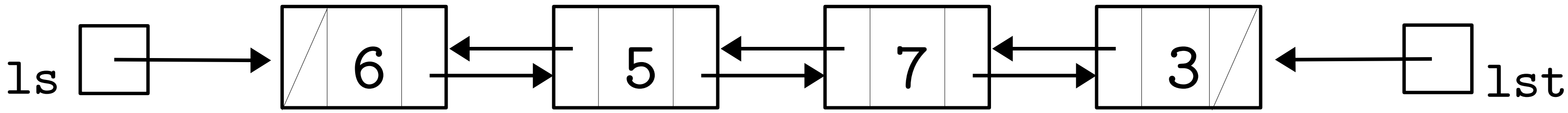
Perform the following operations in sequence and draw the new lists:

e)
```
ptr=list1
while ptr.next!=None:
  ptr.data = 'm'
  ptr=ptr.next
list2.data = ptr.data
ptr.data = 'o'
```

Further notions: doubly linked lists

Accessing and traversing lists from the left can be inefficient (e.g. for adding elements at the end of lists).

A solution to this is to have two pointers in each node: one pointing left, and one pointing right. This gives *doubly linked lists*.



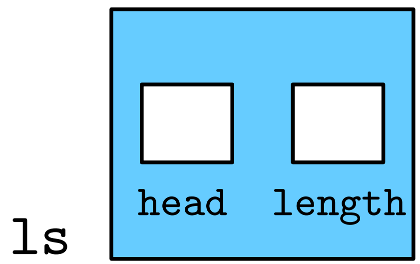
In this case, a node contains two pointer variables:

```
class DNode:
    def __init__(self, d, n, p):
        self.data = d
        self.next = n
        self.prev = p
```

Time for objects

So far we have been somewhat sloppy and not presented linked lists as objects (we only presented nodes as objects).

We describe a linked list as an object containing its first node, which we call the list's **head**, and a variable storing the length of the list:



Some notes:

- `head` is a variable of type `Node`
- when `ls` is the empty list, we set its `head` to `None`
- we can expand the above e.g. by storing not only the head of the linked list, but also the last element, the least element, etc.

A LinkedList class

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.length = 0

    def search(self, d):
        i = 0
        ptr = self.head
        while ptr != None:
            if ptr.data == d:
                return i
            ptr = ptr.next
            i += 1
        return -1

    def append(self, d):
        if self.head == None:
            self.head = Node(d,None)
        else:
            ptr = self.head
            while ptr.next != None:
                ptr = ptr.next
            ptr.next = Node(d,None)
        self.length += 1
```

```
def insert(self, i, d):
    if self.head == None or i == 0:
        self.head = Node(d,self.head)
    else:
        ptr = self.head
        while i>1 and ptr.next != None:
            ptr = ptr.next
            i -= 1
        ptr.next = Node(d,ptr.next)
    self.length += 1

def remove(self, i): # removes/returns i-th element
    if self.head == None:
        return None
    if i == 0:
        val = self.head.data
        self.head = self.head.next
        self.length -= 1
        return val
    else:
        ptr = self.head
        while i>1 and ptr.next != None:
            ptr = ptr.next
            i -= 1
        if i == 1:
            val = ptr.next.data
            ptr.next = ptr.next.next
            self.length -= 1
            return val
        return None
```

Application 1: stacks

```
class Stack:

    def __init__(self):
        self.inList = LinkedList()

    def size(self):
        return self.inList.length

    def push(self, e):
        self.inList.insert(0,e)

    def pop(self):
        return self.inList.remove(0)
```

Application 2: queues

```
class Queue:

    def __init__(self):
        self.inList = LinkedList()

    def size(self):
        return self.inList.length

    def enq(self, e):
        self.inList.append(e)

    def deq(self):
        return self.inList.remove(0)
```

An array list implementation using linked lists

```
class ArrayList2:
    def __init__(self):
        self.head = None
        self.length = 0

    def get(self, i):
        ptr = self.head
        while i>0 and ptr != None:
            ptr = ptr.next
            i -= 1
        return ptr.data

    def set(self, i, d):
        ptr = self.head
        while i>0 and ptr != None:
            ptr = ptr.next
            i -= 1
        ptr.data = d

    def append(self, d):
        if self.head == None:
            self.head = Node(d,None)
        else:
            ptr = self.head
            while ptr.next != None:
                ptr = ptr.next
            ptr.next = Node(d,None)
        self.length += 1
        return None
```

```
    def insert(self, i, d):
        if self.head == None or i == 0:
            self.head = Node(d,self.head)
        else:
            ptr = self.head
            while i>1 and ptr.next != None:
                ptr = ptr.next
                i -= 1
            ptr.next = Node(d,ptr.next)
        self.length += 1

    def remove(self, i):
        if self.head == None:
            return None
        if i == 0:
            val = self.head.data
            self.head = self.head.next
            self.length -= 1
            return val
        else:
            ptr = self.head
            while i>1 and ptr.next != None:
                ptr = ptr.next
                i -= 1
            if i == 1:
                val = ptr.next.data
                ptr.next = ptr.next.next
                self.length -= 1
                return val
```

linked list vs array and count

We have now seen two data structures that are linear (store elements in a line) and dynamic in size: linked lists and array-and-count's

array and count	
easy extension of arrays	use up to twice the space needed
read/write arbitrary element in O(1)	insert/remove arbitrary element takes O(n)
append an element in O(1) (most times)	insert/remove element in head pos. takes O(n)
if sorted, search of elements in O(log n) (via binary search)	cannot take sub-arrays (can only make sub-copies), also merging arrays takes O(n)

linked list	
use exactly the space needed	read/write arbitrary element is O(n)
read/write and inserting/removing element in head pos. takes O(1)	insert/remove arbitrary element takes O(n)
append an element in O(1) *	even if sorted, search of elements takes O(n)
can take sub-lists, merging linked lists takes O(1) *	appending, merging takes O(n)

* if we have pointer to last node in list

How to form sublists

Suppose that `ls` a linked list.

How can we create the sublist starting from the third element of `ls`?

We can use this function inside `LinkedList`:

```
def sublist(self, i):
    ptr = self.head
    ls = LinkedList()
    ls.length = self.length
    while ptr != None and i>0:
        ptr = ptr.next
        i -= 1
        ls.length -= 1
    ls.head = ptr
    return ls
```

Note: in an array-and-count implementation of lists this operation would be much more involved. Similarly for merging two lists.

Summary

Linked lists are an important data structure, and can be used in order to implement data types such array lists, queues and stacks.

They are very efficient for adding/removing elements from arbitrary positions, but not as good for finding and indexing their elements.

They are good choice for implementing lists that are sublisted often as they allow for a space efficient implementation where the same nodes are used in several places.

Exercises

1. Implement a function

   ```
   def removeVal(self, d)
   ```

   inside `LinkedList` that searches for the value `d` and removes its first occurrence in the linked list (if appears in it).

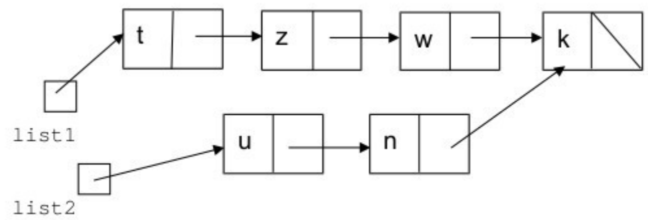
2. Using the code for `LinkedList`, implement a class `LinkedList2` that represents doubly-linked lists. `LinkedList2` objects should have a pointer to their head and the tail nodes, which should be from the class:

   ```
   class DNode:
       def __init__(self, d, n, p):
       self.data = d
           self.next = n
           self.prev = p
   ```

3. Let `ls` be the linked list `[2,3,4,5,42]`. Do the following operations in sequence:

   ```
   ls.remove(3)
   ls.insert(2,24)
   ls.append(23)
   print(ls.remove(4))
   ```

4. Consider this initial pointer game:



and perform the following operations:

a)
```
list1.next = list2.next
ptr = list1
while ptr!=None:
      ptr.data = 's'
      ptr=ptr.next
```

b)
```
ptr=list2
while ptr.next!=None: ptr=ptr.next
ptr.data = 'e'
ptr.next = list1
```

c)
```
ptr=list1
while ptr.next!=None: ptr=ptr.next
ptr.next = list2
count = 1
ptr=list2
while ptr!=None:
      list1 = Node('c',list1)
      count += 1
      ptr=ptr.next
if count%2 == 0:
      list1.data = 'a'
else:
      list1.data = 'b'
```

Exercises – useful code

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.length = 0

    def search(self, d):
        i = 0
        ptr = self.head
        while ptr != None:
            if ptr.data == d:
                return i
            ptr = ptr.next
            i += 1
        return -1

    def append(self, d):
        if self.head == None:
            self.head = Node(d,None)
        else:
            ptr = self.head
            while ptr.next != None:
                ptr = ptr.next
            ptr.next = Node(d,None)
        self.length += 1
```

```
    def insert(self, i, d):
        if self.head == None or i == 0:
            self.head = Node(d,self.head)
        else:
            ptr = self.head
            while i>1 and ptr.next != None:
                ptr = ptr.next
                i -= 1
            ptr.next = Node(d,ptr.next)
        self.length += 1

    def remove(self, i): # removes/returns i-th element
        if self.head == None:
            return None
        if i == 0:
            val = self.head.data
            self.head = self.head.next
            self.length -= 1
            return val
        else:
            ptr = self.head
            while i>1 and ptr.next != None:
                ptr = ptr.next
                i -= 1
            if i == 1:
                val = ptr.next.data
                ptr.next = ptr.next.next
                self.length -= 1
                return val
            return None
```

Exercises – solutions

1. The simplest way to implement it is using the other functions of the `LinkedList` class:

```
def removeVal(self, d):
    i = self.search(d)
    if i!=-1:
        self.remove(i)
```

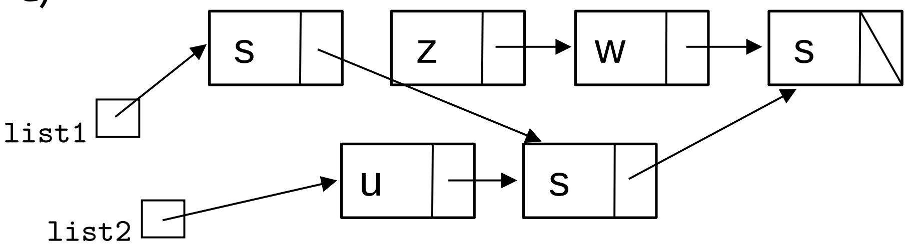
   This has the advantage that we reuse code that we have already implemented (and, in theory, checked). It has the disadvantage that it traverses the list twice: one for searching the value `d`, and one for removing element `i`.
   An alternative solution, which traverses the list once, is given on this lecture's notebook file.

3. Here are the operations:

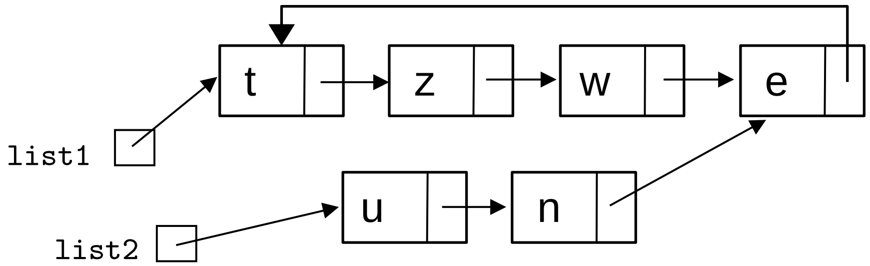
```
ls.remove(3)         # ls = [2,3,4,42]
ls.insert(2,24)      # ls = [2,3,24,4,42]
ls.append(23)        # ls = [2,3,24,4,42,23]
print(ls.remove(4))  # ls = [2,3,24,4,23]
```

4. Here is the outcome in each case:

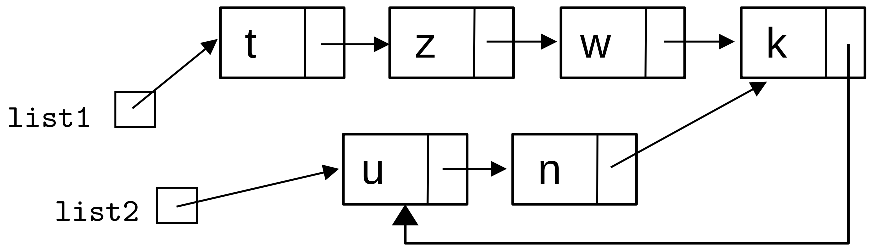
a)



b)



c) in this case, the first 3 lines create a cycle in the lists:



Then, we set `count` to 1 and let `ptr` point to the first element of `list2` (i.e. u) and go into this while loop:

```
while ptr!=None:
    list1 = Node('c',list1)
    count += 1
    ptr=ptr.next
```

From there on, we will never exit this loop as `ptr` is going to be rotating between u, n and k forever. So, we will keep extending `list1` at the front with new nodes (with value c) and increasing `count`.
In particular, we will not reach the if-the-else clause after the while loop.