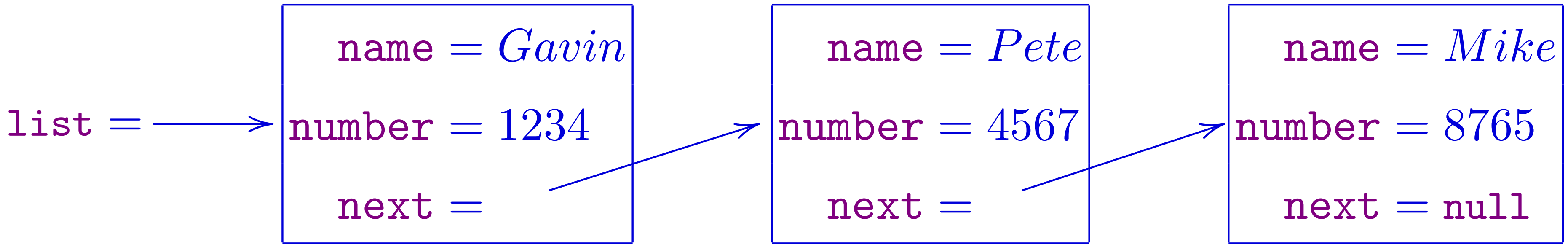
# IP Lecture 14: Programming with linked lists

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

# `LinkedListBook` abstraction function and DTI




Recall that $L(a)$ is defined to be the list of nodes reachable from $a$ by following `next` references.

The abstraction function is

$$\textbf{Abs:}\ \ book = \{n.\texttt{name} \to n.\texttt{number} \mid n \in L(\texttt{list})\}$$

Our intention is that the lists are finite, which implies that they are acyclic; and that names are not repeated:

$$\textbf{DTI:}\ \ L(\texttt{list}) \text{ is finite, and the names in } L(\texttt{list}) \text{ are distinct}$$

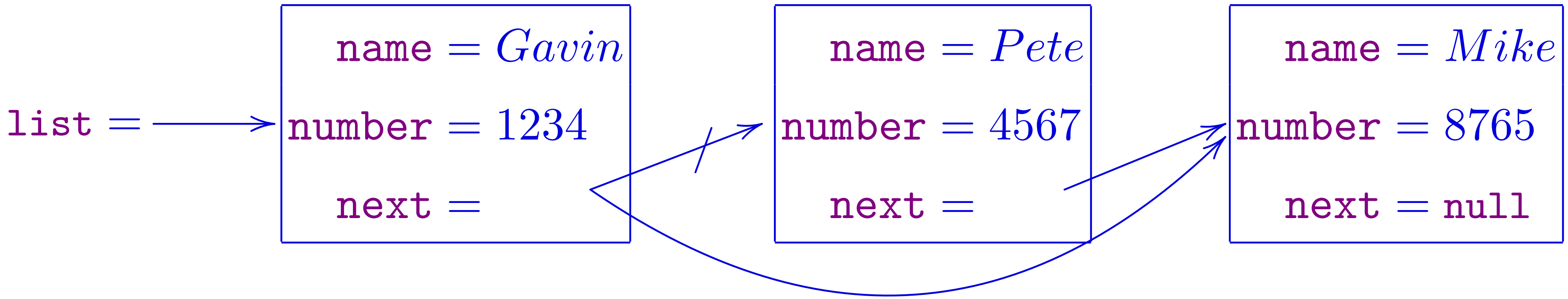
# find and store

```
/** return (n in L(list) && n.name=name)
  *        or n=null if no such Node exists */
private def find(name: String) : Node = {
  var n = list
  // Inv: for all n1 in L(list,n), n1.name != name
  while(n != null && n.name != name) n = n.next
  n
}
```

```
/** Add the maplet name -> number to the mapping */
def store(name: String, number: String) = {
  val n = find(name);
  if(n==null){
    val n1 = new Node(name, number, list)
    list = n1
  }
  else n.number = number
}
```

# Deleting from a linked list

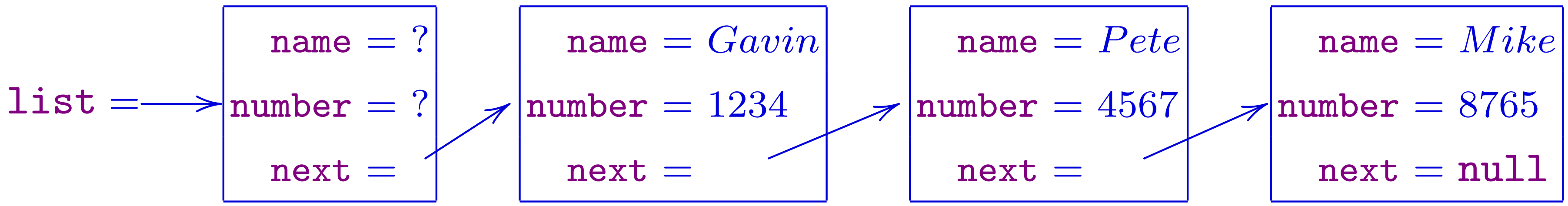
To delete an entry from a linked list we have to find the node `n` before that entry, and short circuit it, changing `n.next` to point to the following node.




But deleting the first entry in the list has to be treated as a special case. This is messy.

# Using a dummy header node

A better technique is to use a dummy header node: a node that does not contain any real data, but whose `next` field points to the first proper node.



# Linked lists with a dummy header

We need to initialise the linked list appropriately

```
class LinkedListBook extends Book{
  private var list = new Node("?", "?", null)
  // Abs: book = { n.name -> n.number | n <- L(list.next) }
  // DTI: L(list) is finite and
  //      the names in L(list.next) are distinct
  ...
}
```

Note that now *book* is formed from the names and numbers in nodes starting from `list.next`.

# `find`

We arrange for `find` to find the node before the one containing the given name, or the last node if there is no such. That's what we'll want in order to implement `delete`.

```
/** Return the node before the one containing name.
  * Post: book = book_0 && returns n s.t. n in L(list) &&
  * (n.next.name=name or n.next=null if no such Node exists)*/
private def find(name:String) : Node = {
  var n = list
  // Invariant: name does not appear in the nodes up to and
  // including n; i.e.,
  // for all n1 in L(list.next, n.next), n1.name != name
  while(n.next != null && n.next.name != name) n = n.next
  n
}
```

# isInBook and recall

`isInBook` and `recall` are easily adapted to use the new version of `find`

```
/** Is name in the book? */
def isInBook(name: String): Boolean = find(name).next != null

/** Return the number stored against name */
def recall(name: String) : String = {
  val n = find(name); assert(n.next != null); n.next.number
}
```

## store

For `store`, if the name doesn't already exist, we store the new data in the old header cell:

```
/** Add the maplet name -> number to the mapping */
def store(name: String, number: String) = {
  val n = find(name)
  if(n.next == null){ // store new info in current list header
    list.name = name; list.number = number
    list = new Node("?", "?", list)
  }
  else n.next.number = number
}
```

Alternatively, we could have created a new cell to hold the new data:

```
  if(n.next == null){
    val n1 = new Node(name, number, list.next); list.next = n1
  }
```

## `delete`

To delete a name and number, we find the node `n` before the node containing that name, and then short-circuit that node. Also, we return a `Boolean` indicating whether the name was found in the list (following the convention of the Scala API).

```
/** Delete the number stored against name (if it exists);
  * return true if the name existed. */
def delete(name: String) : Boolean = {
  val n = find(name)
  if(n.next != null){ n.next = n.next.next; true }
  else false
}
```

# Companion objects

So far, we have considered the class `Node` as being defined at the top level. In some ways, this is the easiest approach. However, it means that `Node` is globally visible. This is undesirable, because we would like to hide the implementation, i.e. we would like to make the class `Node` `private`.

We could try putting the definition of `Node` inside the `LinkedListBook` class definition:

```
class LinkedListBook extends Book{
  private class Node(var name: String, var number: String, var next: Node)
  ...
}
```

However, if we have two different `LinkedListBook` objects, `l1` and `l2`, then their `Node`s will be of different types, `l1.Node` and `l2.Node`, which will make it difficult to compare them.

## Companion objects

Definitions inside the `LinkedListBook` class should apply to particular objects formed from that class template.

By contrast, if we want definitions to apply to all such `LinkedListBook` objects, they should be inside the companion object:

```
object LinkedListBook{
  private class Node(var name: String, var number: String, var next: Node)
}
```

The `LinkedListBook` object is known as the companion object of the `LinkedListBook` class, and defines types/values/operations that relate to all objects of the type, as opposed to those that operate on a single object of the type.

# Companion objects

`Node` is defined as `private` inside the companion object, but it is still visible within the `LinkedListBook` class. References to `Node` need to be written as `LinkedListBook.Node`, e.g.:

```
private var list : LinkedListBook.Node = null
...
val n1 = new LinkedListBook.Node(name, number, list)
```

# Companion objects

- A class defines types/values/operations on a particular object of that class; for example, the `store`, `recall` and `isInBook` operations will be defined in the `LinkedListBook` class because they are operations on a particular object.
- The companion object defines types/values/operations that apply more generally to objects of the class[a]; for example, the `Node` type will be defined in the `LinkedListBook` companion object, because we want the same type to apply to all nodes.

Companion objects and classes should be defined in the same file and have the same name. They can see one another's private fields.

[a] In Java, these would be indicated using the keyword `static`.

# Garbage collection

When we delete a name, the node that holds that name is no longer reachable. At this point, the language implementation recycles that storage automatically, so it can be re-used. This is known as garbage collection. Sometimes it is necessary to explicitly remove a reference to a node (by setting the variable that references the node to `null`) to allow garbage collection to go ahead.

By contrast, old-fashioned languages like C/C++ force the programmer to explicitly manage the de-allocation of memory, which provides extra opportunities for bugs:

- The programmer might fail to deallocate storage that is no longer in use: a memory leak;
- The programmer might deallocate storage that is still in use.

# Keeping the names in order

An alternative design would keep the list ordered in alphabetical order of the names. This would make it easy to print out the names in alphabetical order. It would also mean that if we search for a name `name` and get to a node `n` with `n.name > name`, we know `name` is not in the list.

Exercise: implement a class using this idea.

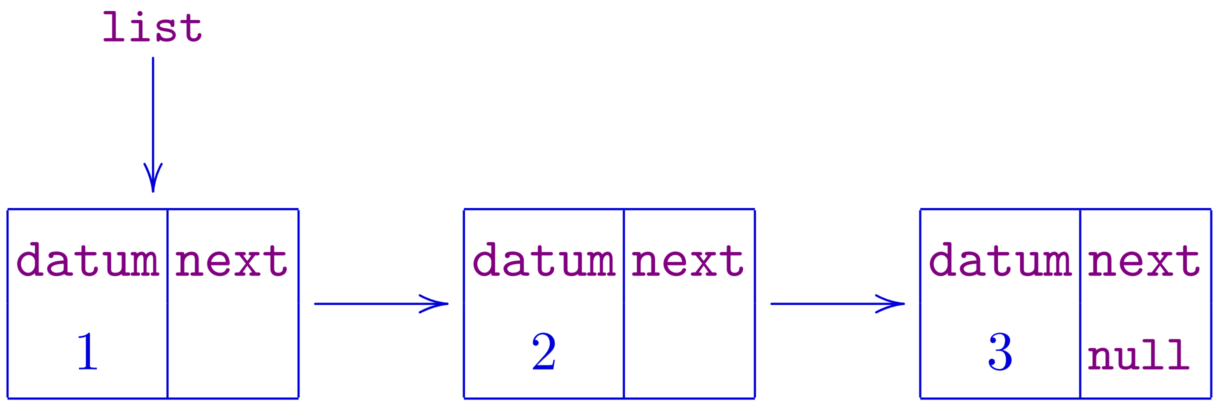
# Why not `case` classes?

We defined `Node` not as a `case` class. The reason is that we want to be able to write `n1 == n2` to test if `n1` and `n2` are the same node (i.e. if `n1` and `n2` are equal as references). However, if we made `Node` a `case` class, the Scala compiler would create a default implementation of `==`, such that `n1 == n2` would be equivalent to

```
n1.name == n2.name && n1.number == n2.number && n1.next == n2.next
```

In particular, the last clause does a similar check on the `next` nodes, so this might traverse the whole list — causing particular difficulties if the lists contain loops.

# Linked list variants




Linked lists are generally useful for a variety of abstract datatypes:

- dynamically sized container
- queue (first-in first-out)
- stack
- double-ended queue (deque) ...

Depending on the datatype required and the cost of operations we might want to modify the structure.

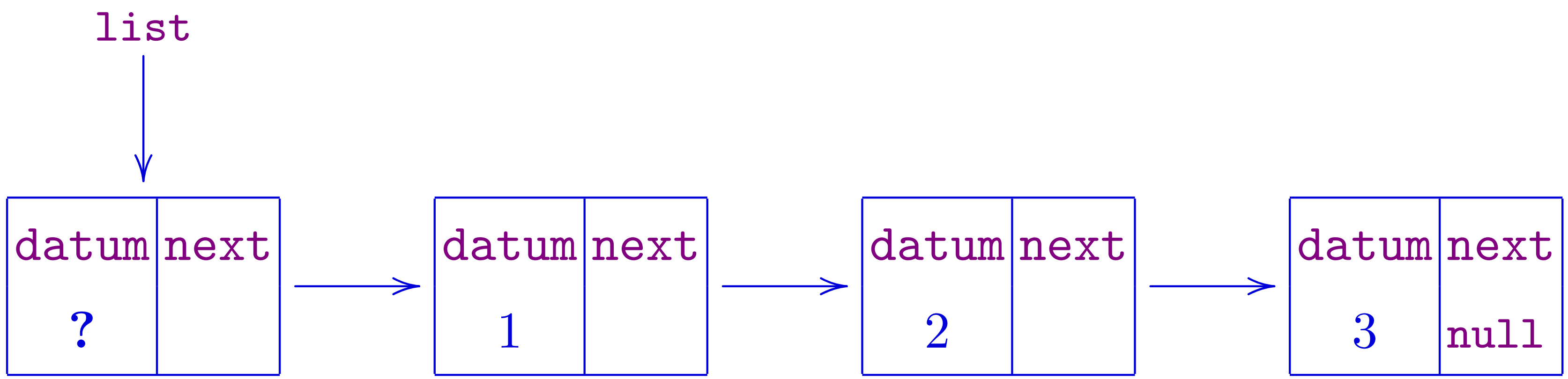
# Dummy header

Motivation:

- Recall that add/delete of the first node in linked list are special cases because we need to update `list` reference.

Solution:

- Maintain an extra node at the beginning of the list which is not used to store real data.



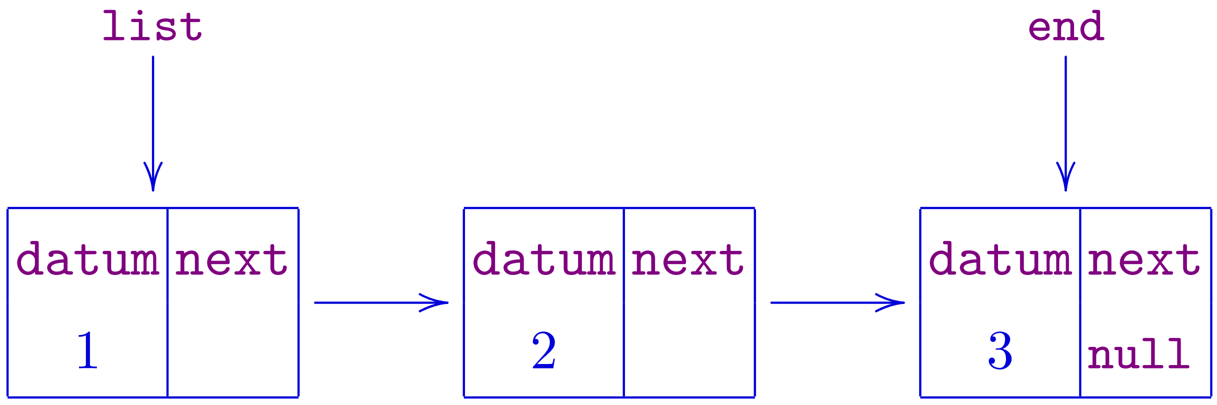
# "Tailed" linked list

Motivation:

- The last node in singly linked list takes the longest time to reach.
- Some datatypes (queue) require the addition of a new item to the end of list—very inefficient.

Solution:

- Keep an extra reference to the last entry of the list.



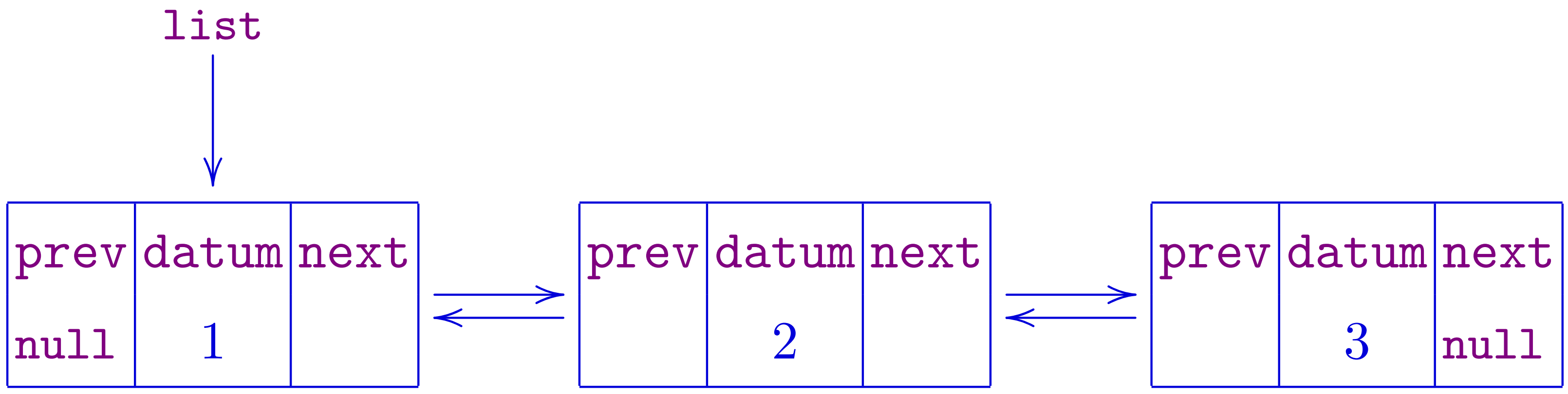
# Doubly linked list

Motivation:

- Singly linked lists can only be navigated in one direction.
- The `delete` operation requires that we know the predecessor of the node to be removed.

Solution:

- All nodes contain a `prev` reference (and a `next` reference).
- Note: `delete` and `add` have extra housekeeping.



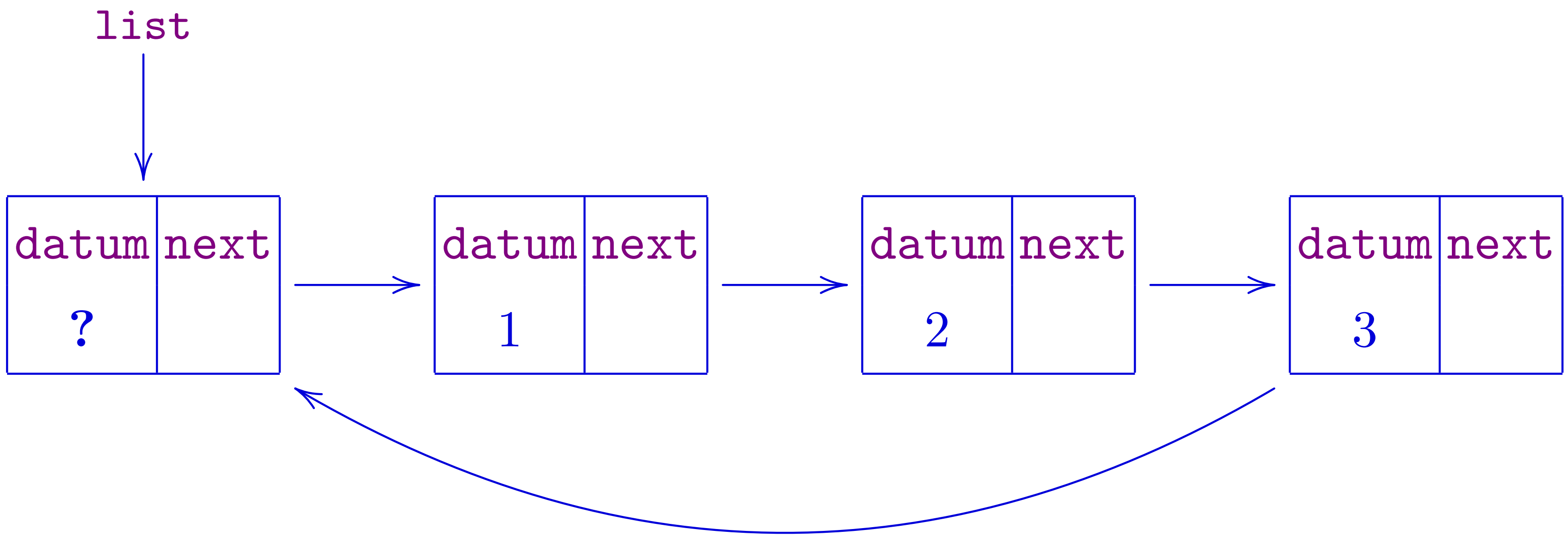
# Circular linked list

Motivation:

- Want to be able to navigate structure multiple times.
- Want to be able to rotate the data.

Solution:

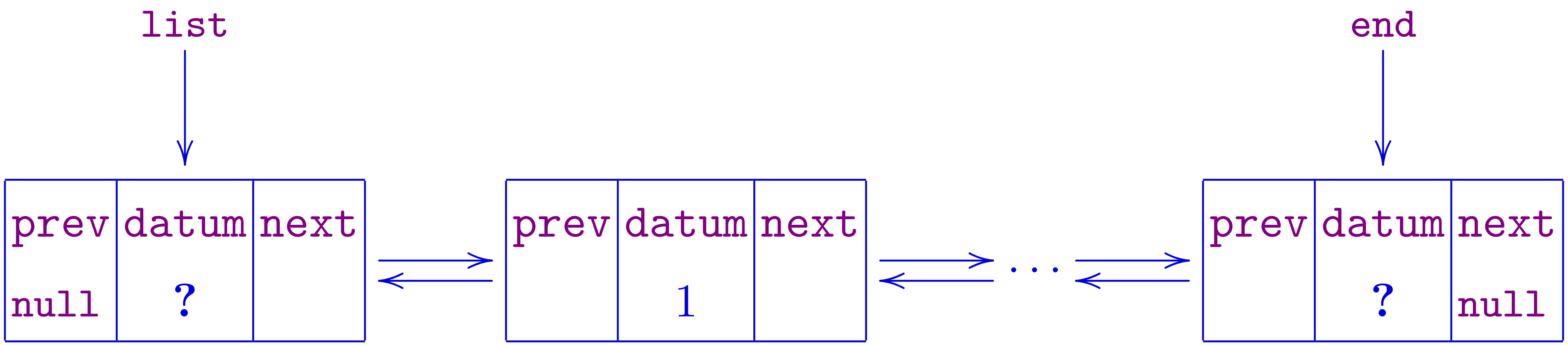
- Connect the `last` node back to the head.
- `while(n!=null)` becomes `while(n!=list)`



# Combinations

A doubly-ended queue (deque) requires $O(1)$ operations to `add` or `delete` at either end. This is could be done with

- A doubly linked list with dummy nodes




- or (with thought) a circular linked list

## Summary

- Abstraction function;
- Dummy header nodes;
- Companion objects;
- Garbage collection.
- Doubly linked lists and circular lists;
- Next time: Programming with datatypes.