Databases and Algorithms

Array and Linked List

Instructor: Rossano Schifanella

@SDS

Objectives

- After completing this chapter, you will be able to:
- Recognize different categories of collections and the operations on them
- Understand the difference between an abstract data type and the concrete data structures used to implement it
- Differentiate between arrays and linked lists.

Objectives (continued)

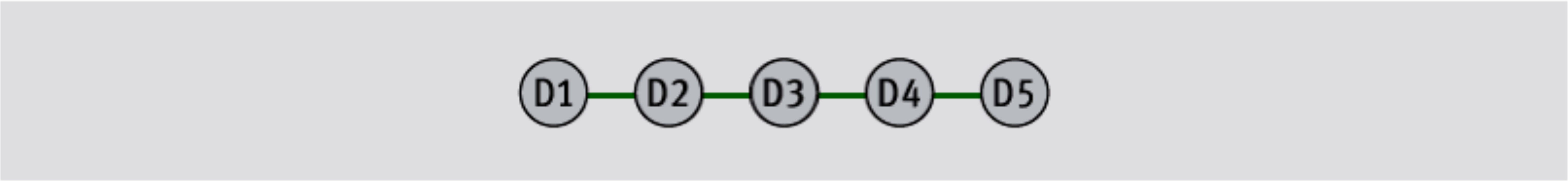
- Describe the space/time trade-offs for users of arrays
- Perform basic operations, such as traversals, insertions, and removals, on linked structures
- Explain the space/time trade-offs of arrays and linked structures in terms of the memory models that underlie these data structures

Overview of Collections

- Collection: Group of items that we want to treat as conceptual unit
- Examples:
  - Lists, strings, stacks, queues, binary search trees, heaps, graphs, dictionaries, sets, and bags
- Can be homogeneous or heterogeneous
  - Lists are heterogeneous in Python
- Four main categories:
  - Linear, hierarchical, graph, and unordered

Linear Collections

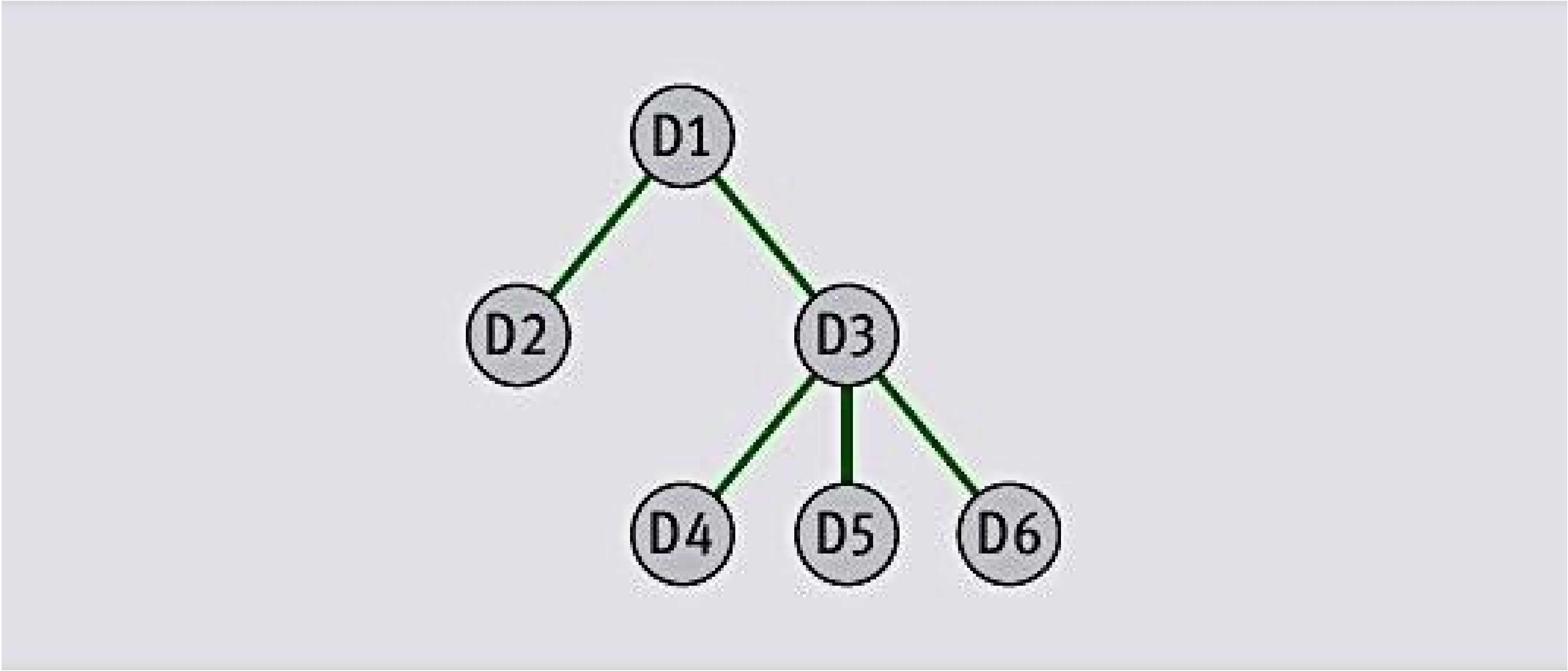
- Ordered by position



- Everyday examples:
  - Grocery lists
  - Stacks of dinner plates
  - A line of customers waiting at a bank

Hierarchical Collections

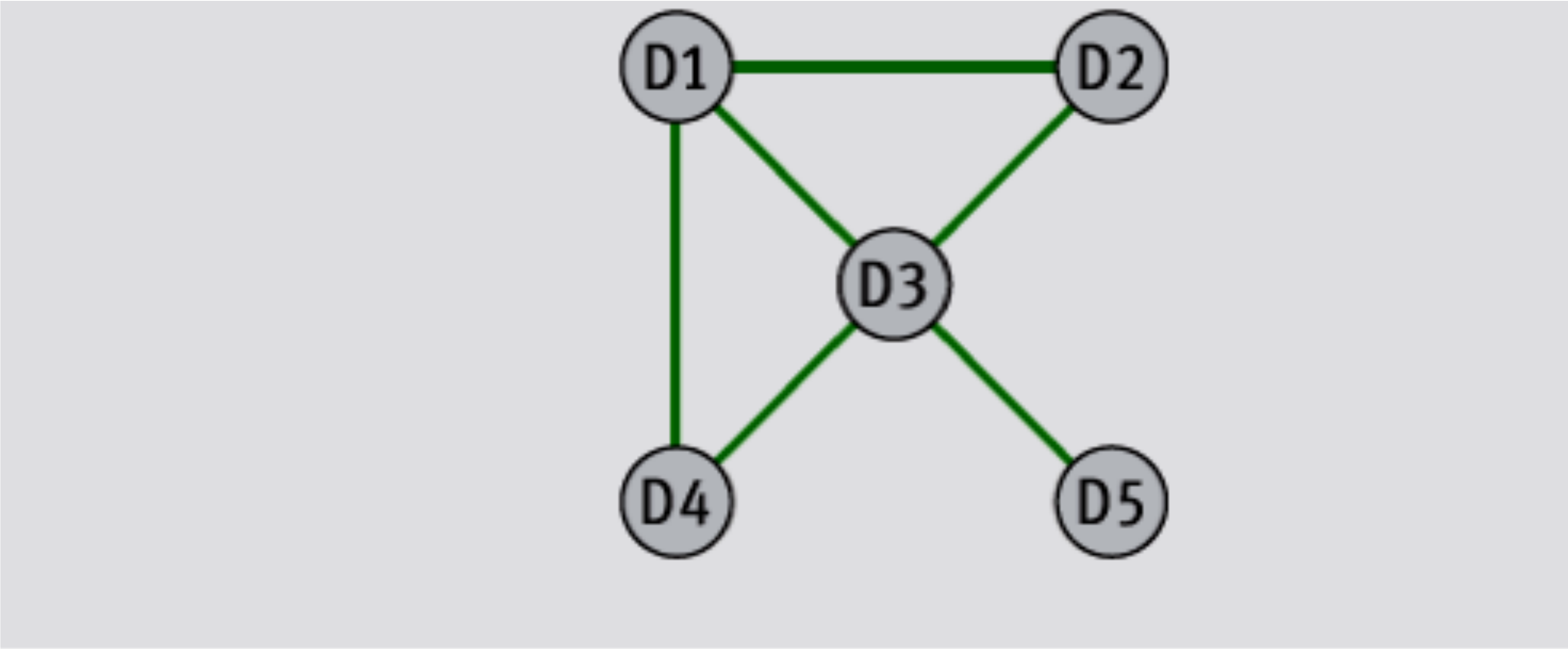
- Structure reminiscent of an upside-down tree



  - D3's parent is D1; its children are D4, D5, and D6
- Examples: a file directory system, a company's organizational tree, a book's table of contents

Graph Collections

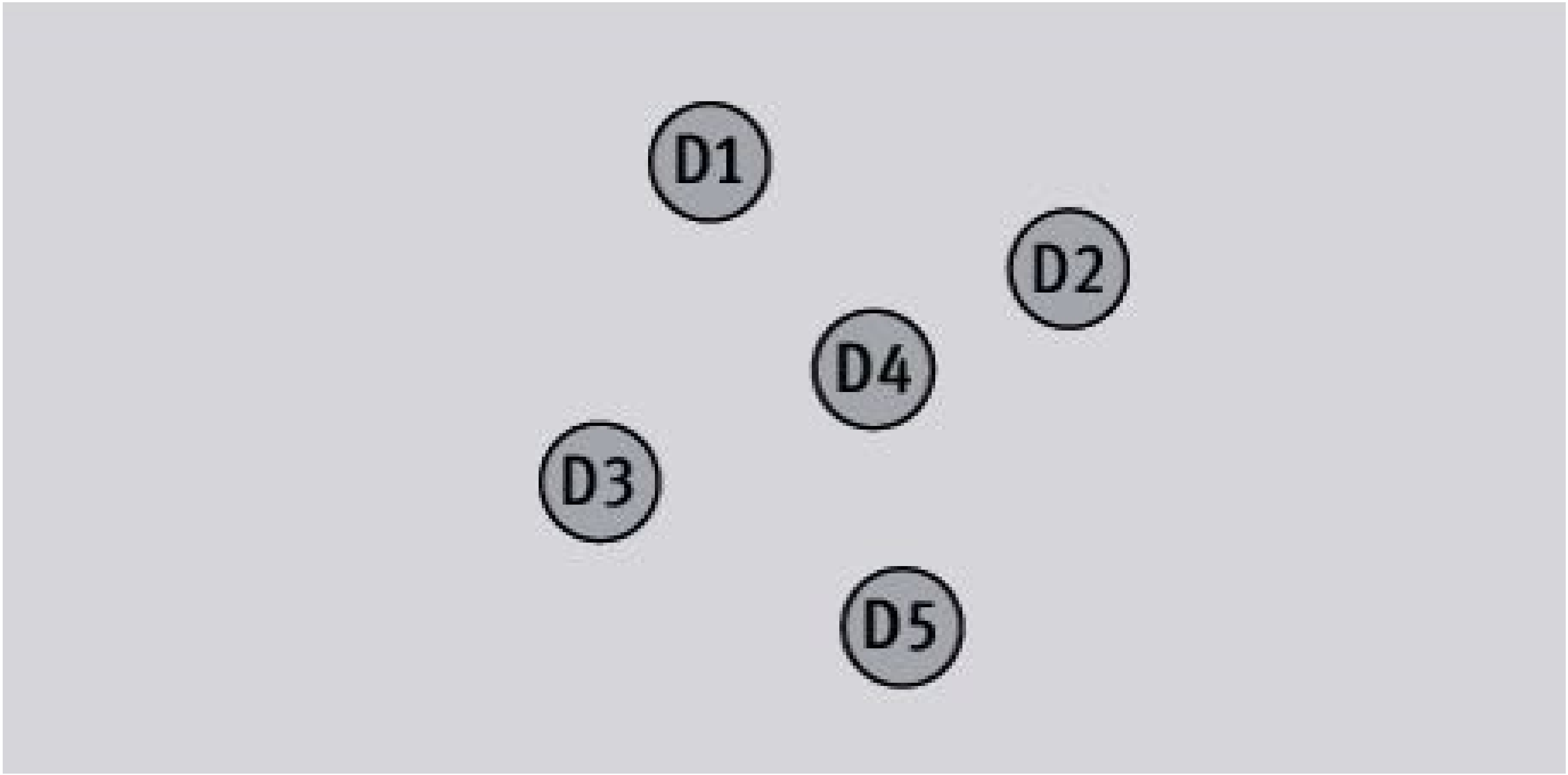
- Graph: Collection in which each data item can have many predecessors and many successors



  - D3's neighbors are its predecessors and successors
- Examples: Maps of airline routes between cities; electrical wiring diagrams for buildings

Unordered Collections

- Items are not in any particular order
  - One cannot meaningfully speak of an item's predecessor or successor



- Example: Bag of marbles

Operations on Collections

CATEGORY OF OPERATION	DESCRIPTION
Search and retrieval	These operations search a collection for a given target item or for an item at a given position. If the item is found, either it or its position is returned. If the item is not found, a distinguishing value, such as `None` or –1, is returned.
Removal	This operation deletes a given item or the item at a given position.
Insertion	This operation adds an item to a collection, usually at a particular position within the collection.
Replacement	This operation combines removal and insertion into a single operation.
Traversal	This operation visits each item in a collection. Depending on the type of collection, the order in which the items are visited can vary. During a traversal, items can be accessed or modified. Collections that can be traversed with Python's `for` loop are said to be **iterable**.

CATEGORY OF OPERATION	DESCRIPTION
Test for equality	This operation tests two items to see if they are equal. If items can be tested for equality, then the collections containing them can also be tested for equality. To be equal, two collections must contain equal items at corresponding positions. For unordered collections, of course, the requirement of corresponding positions can be ignored. Some collections, such as strings, also can be tested for their position in a natural ordering using the comparisons less than and greater than.
Determine the size	This operation determines the size of a collection—the number of items it contains. Some collections also have a maximum capacity, or number of places available for storing items. An egg carton is a familiar example of a container with a maximum capacity.
Cloning	This operation creates a copy of an existing collection. The copy usually shares the same items as the original, a feat that is impossible in the real world. In the real world, a copy of a bag of marbles could not contain the same marbles as the original bag, given a marble's inability to be in two places at once. The rules of cyberspace are more flexible, however, and there are many situations in which we make these strange copies of collections. What we are copying is the structure of the collection, not the elements it contains. It is possible, however, and sometimes useful to produce a **deep copy** of a collection in which both the structure and the items are copied.

Abstraction and Abstract Data Types

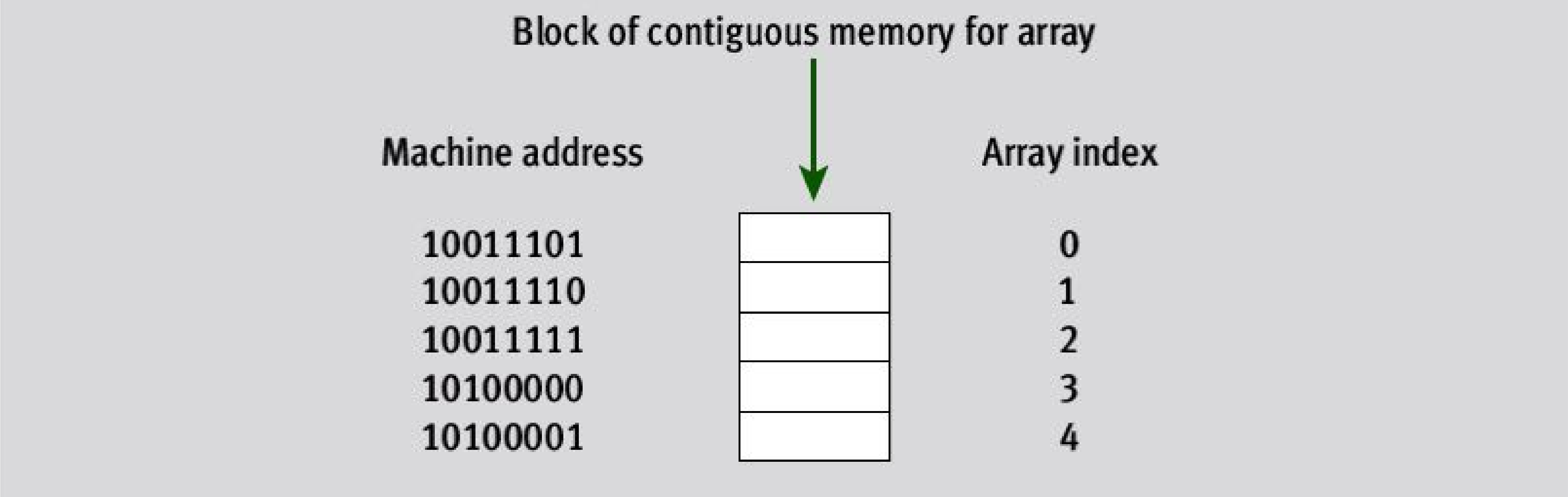
- To a user, a collection is an abstraction
- In CS, collections are abstract data types (ADTs)
  - ADT users are concerned with learning its interface
  - Developers are concerned with implementing their behavior in the most efficient manner possible
- In Python, methods are the smallest unit of abstraction, classes are the next in size, and modules are the largest
- We will implement ADTs as classes or sets of related classes in modules

Data Structures for Implementing Collections: Arrays

- “Data structure” and “concrete data type” refer to the internal representation of an ADT’s data
- The two data structures most often used to implement collections in most programming languages are arrays and linked structures
  - Different approaches to storing and accessing data in the computer’s memory
  - Different space/time trade-offs in the algorithms that manipulate the collections

Random Access and Contiguous Memory

- Array indexing is a random access operation



Address of an item: base address + offset

Index operation has two steps:

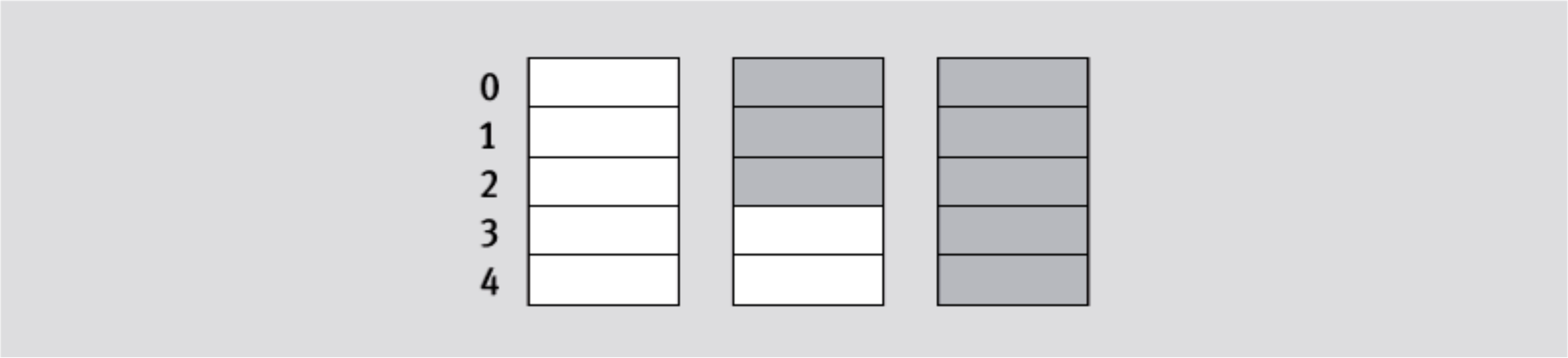
```
Fetch the base address of the array's memory block
Return the result of adding the index * k to this address
```

Static Memory and Dynamic Memory

- Arrays in older languages were **static**
- Modern languages support **dynamic** arrays
- To readjust length of an array at run time:
  - Create an array with a reasonable default size at start-up
  - When it cannot hold more data, create a new, larger array and transfer the data items from the old array
  - When the array seems to be wasting memory, decrease its length in a similar manner
- These adjustments are automatic with Python lists

Physical Size and Logical Size

- The physical size of an array is its total number of array cells
- The logical size of an array is the number of items currently in it



- To avoid reading garbage, must track both sizes

Physical Size and Logical Size (continued)

- In general, the logical and physical size tell us important things about the state of the array:
  - If the logical size is 0, the array is empty
  - Otherwise, at any given time, the index of the last item in the array is the logical size minus 1.
  - If the logical size equals the physical size, there is no more room for data in the array

Array Indexing

- Almost all of the operations on arrays are index based
  - Traversals
  - Insertions
  - Removals
  - Modifications

Operations on Arrays

- We now discuss the implementation of several operations on arrays
- In our examples, we assume the following data settings:

```
DEFAULT_CAPACITY  = 5
logicalSize = 0
a = Array(DEFAULT_CAPACITY)
```

- These operations would be used to define methods for collections that contain arrays

Increasing the Size of an Array

- The resizing process consists of three steps:
  - Create a new, larger array
  - Copy the data from the old array to the new array
  - Reset the old array variable to the new array object

```
if logicalSize == len(a):
    temp = Array(len(a) + 1)           # Create a new array
    for i in xrange(logicalSize):      # Copy data from the old
        temp [i] = a[i]                # array to the new array
    a = temp   # Reset the old array variable to the new array
```

```
temp = Array(len(a) * 2)               # Create new array
```

- To achieve more reasonable time performance, double array size each time you increase its size:

Decreasing the Size of an Array

- This operation occurs in Python's list when a pop results in memory wasted beyond a threshold
- Steps:
  - Create a new, smaller array
  - Copy the data from the old array to the new array
  - Reset the old array variable to the new array object

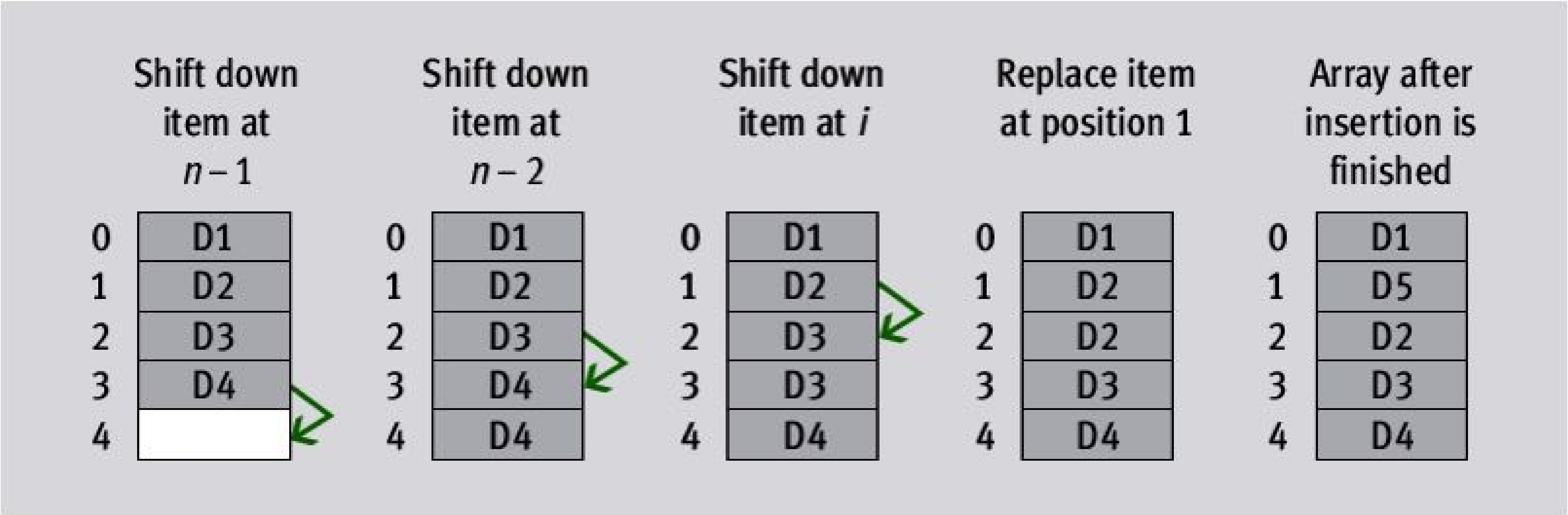
```
if logicalSize <= len(a) / 4 and len(a) > DEFAULT_CAPACITY:
    newSize = max(DEFAULT_CAPACITY, len(a) / 2)
    temp = Array(newSize)           # Create new array
    for i in xrange(logicalSize):   # Copy data from old array
        temp [i] = a [i]            # to new array
    a = temp                        # Reset old array variable to new array
```

Inserting an Item into an Array That Grows

- Programmer must do four things:
  - Check for available space and increase the physical size of the array, if necessary
  - Shift items from logical end of array to target index position down by one
    - To open hole for new item at target index
  - Assign new item to target index position
  - Increment logical size by one

Inserting an Item into an Array That Grows (continued)

Insert D5 in position 1

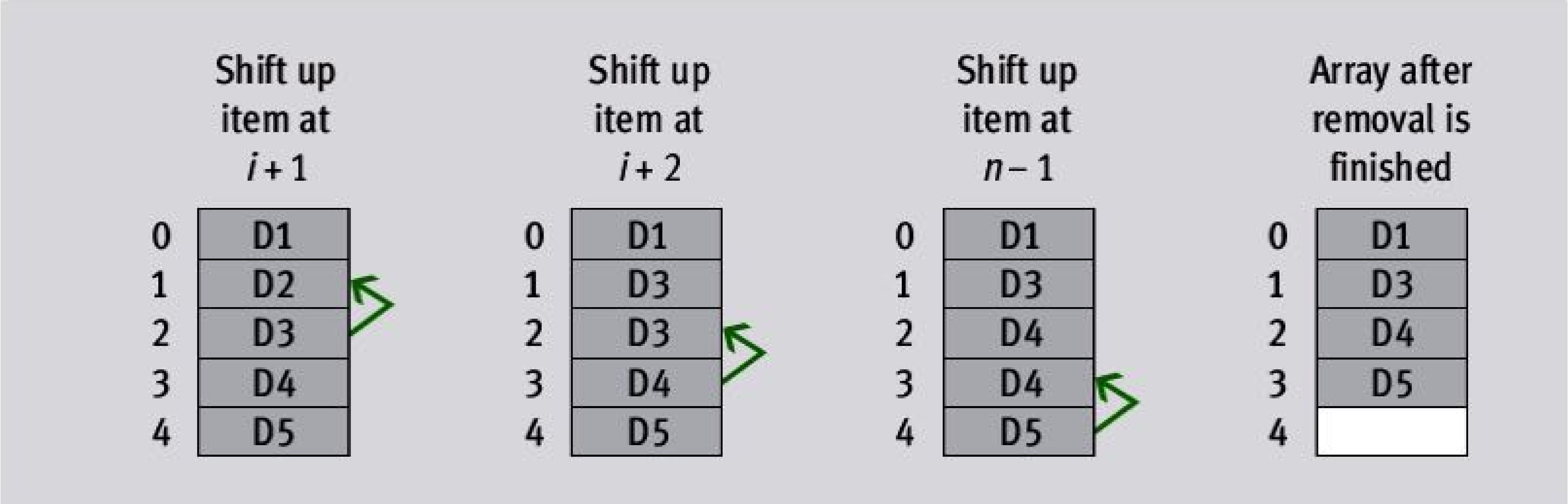


Removing an Item from an Array

- Steps:
  - Shift items from target index position to logical end of array up by one
    - To close hole left by removed item at target index
  - Decrement logical size by one
  - Check for wasted space and decrease physical size of the array, if necessary
- Time performance for shifting items is linear on average; time performance for removal is linear

Removing an Item from an Array (continued)

Remove D2



Complexity Trade-Off: Time, Space, and Arrays

OPERATION	RUNNING TIME
Access at *i*th position	O(1) (best and worst case)
Replacement at *i*th position	O(1) (best and worst case)
Insert at logical end	O(1) (average case)
Remove from logical end	O(1) (average case)
Insert at *i*th position	O(*n*) (average case)
Remove from *i*th position	O(*n*) (average case)
Increase the capacity	O(*n*) (best and worst case)
Decrease the capacity	O(*n*) (best and worst case)

- Memory cost of using an array is its load factor

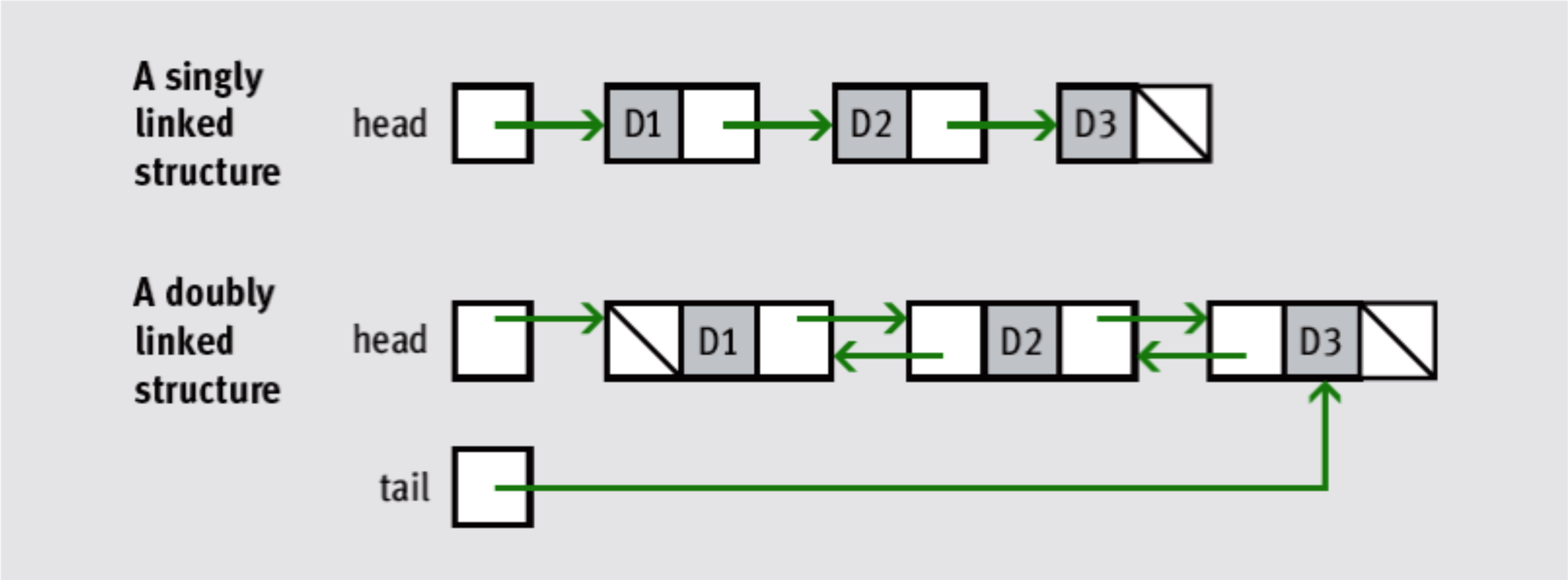
Array in Python

- list
  - http://effbot.org/zone/python-list.htm
  - The list type is a container that holds a number of other objects, in a given order. The list type implements the sequence protocol, and also allows you to add and remove objects from the sequence.
- For efficient numeric array
  - numpy library
  - http://www.numpy.org/

Linked Structures

- After arrays, linked structures are probably the most commonly used data structures in programs
- Like an array, a linked structure is a concrete data type that is used to implement many types of collections, including lists
- We discuss in detail several characteristics that programmers must keep in mind when using linked structures to implement any type of collection

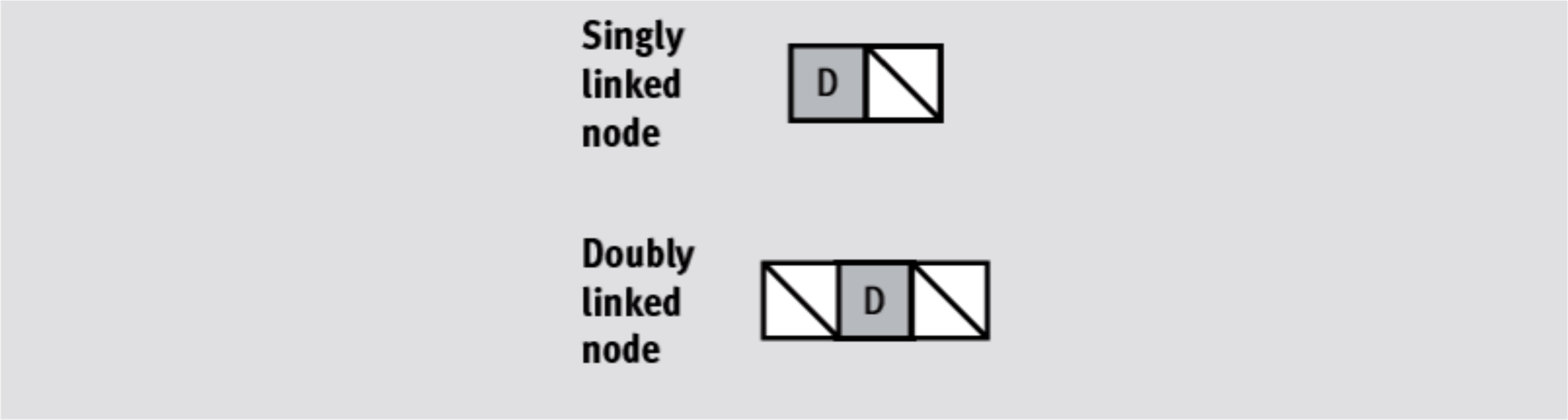
Singly Linked Structures and Doubly Linked Structures



- No random access; must traverse list
- No shifting of items needed for insertion/removal
- Resize at insertion/removal with no memory cost

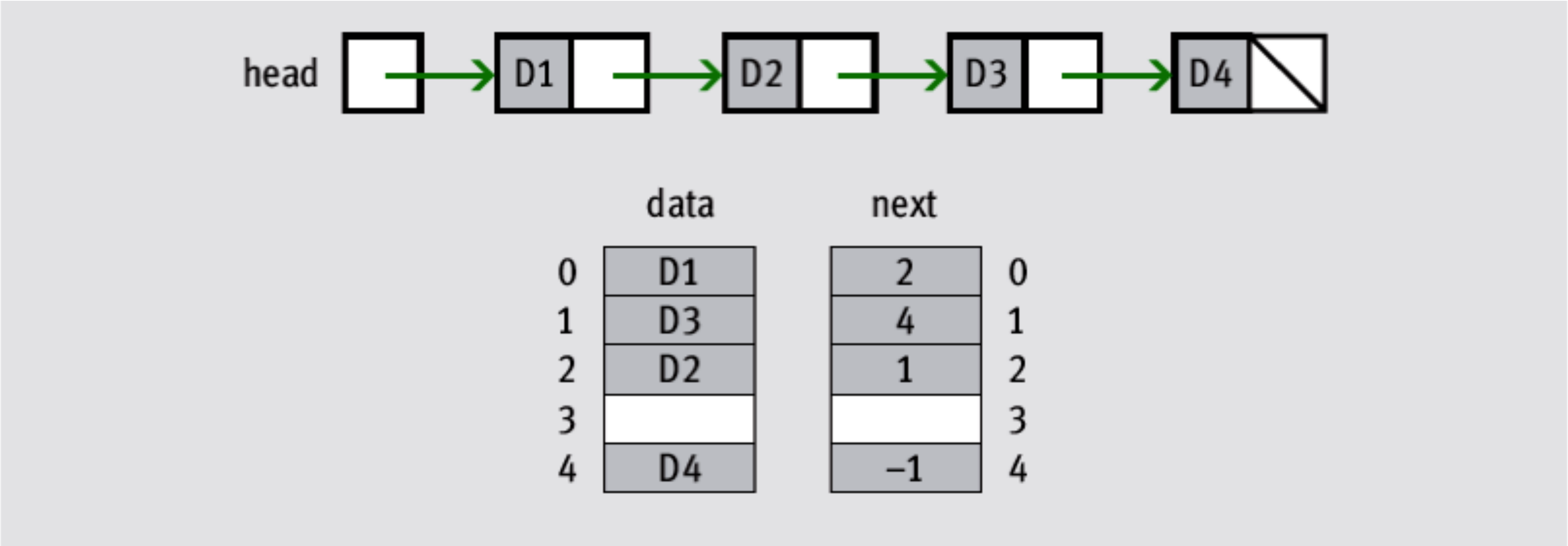
Noncontiguous Memory and Nodes

- A linked structure decouples logical sequence of items in the structure from any ordering in memory
  - Noncontiguous memory representation scheme
- The basic unit of representation in a linked structure is a node:



Noncontiguous Memory and Nodes (continued)

- Depending on the language, you can set up nodes to use noncontiguous memory in several ways:
  - Using two parallel arrays



Noncontiguous Memory and Nodes (continued)

- Ways to set up nodes to use noncontiguous memory (continued):
  - Using pointers (a null or nil represents the empty link as a pointer value)
    - Memory allocated from the object heap
  - Using references to objects (e.g., Python)
    - In Python, None can mean an empty link
    - Automatic garbage collection frees programmer from managing the object heap
- In the discussion that follows, we use the terms link, pointer, and reference interchangeably

Defining a Singly Linked Node Class

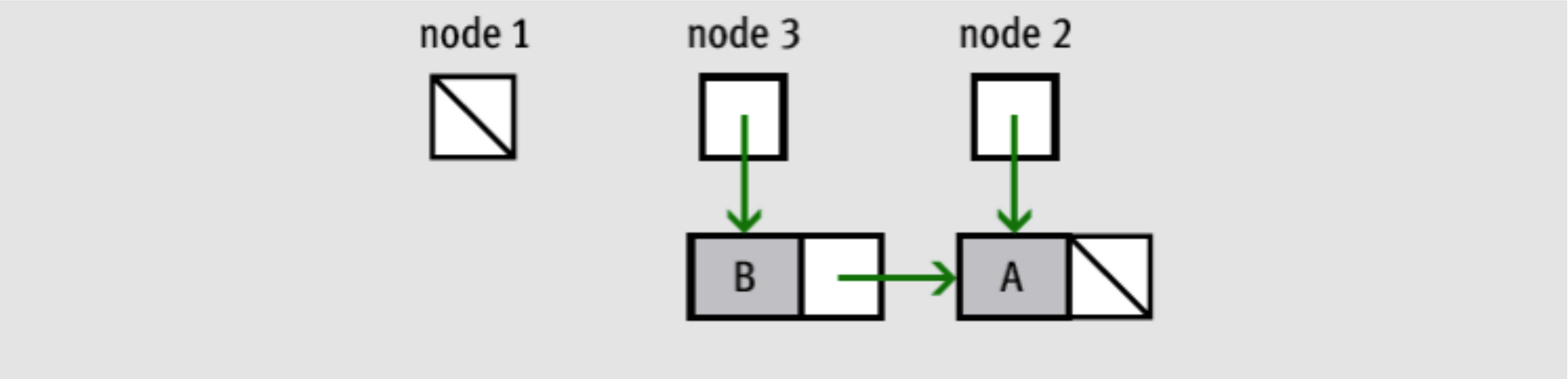
- Node classes are fairly simple
- Flexibility and ease of use are critical
  - Node instance variables are usually referenced without method calls, and constructors allow the user to set a node's link(s) when the node is created
- A singly linked node contains just a data item and a reference to the next node:

```
#Simple Node class without operations
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Using the Singly Linked Node Class

- Node variables are initialized to None or a new Node object

```
#An empty link
node1 = None

#A node with data and an empty link
node2 = Node("A")

#A node with data and a link to another node
node3 = Node("B")
node3.next = node2
```



Using the Singly Linked Node Class (continued)

- Node variables are initialized to None or a new Node object

```
#An empty link
node1 = None

#A node with data and an empty link
node2 = Node("A")

#A node with data and a link to another node
node3 = Node("B")
node3.next = node2
```

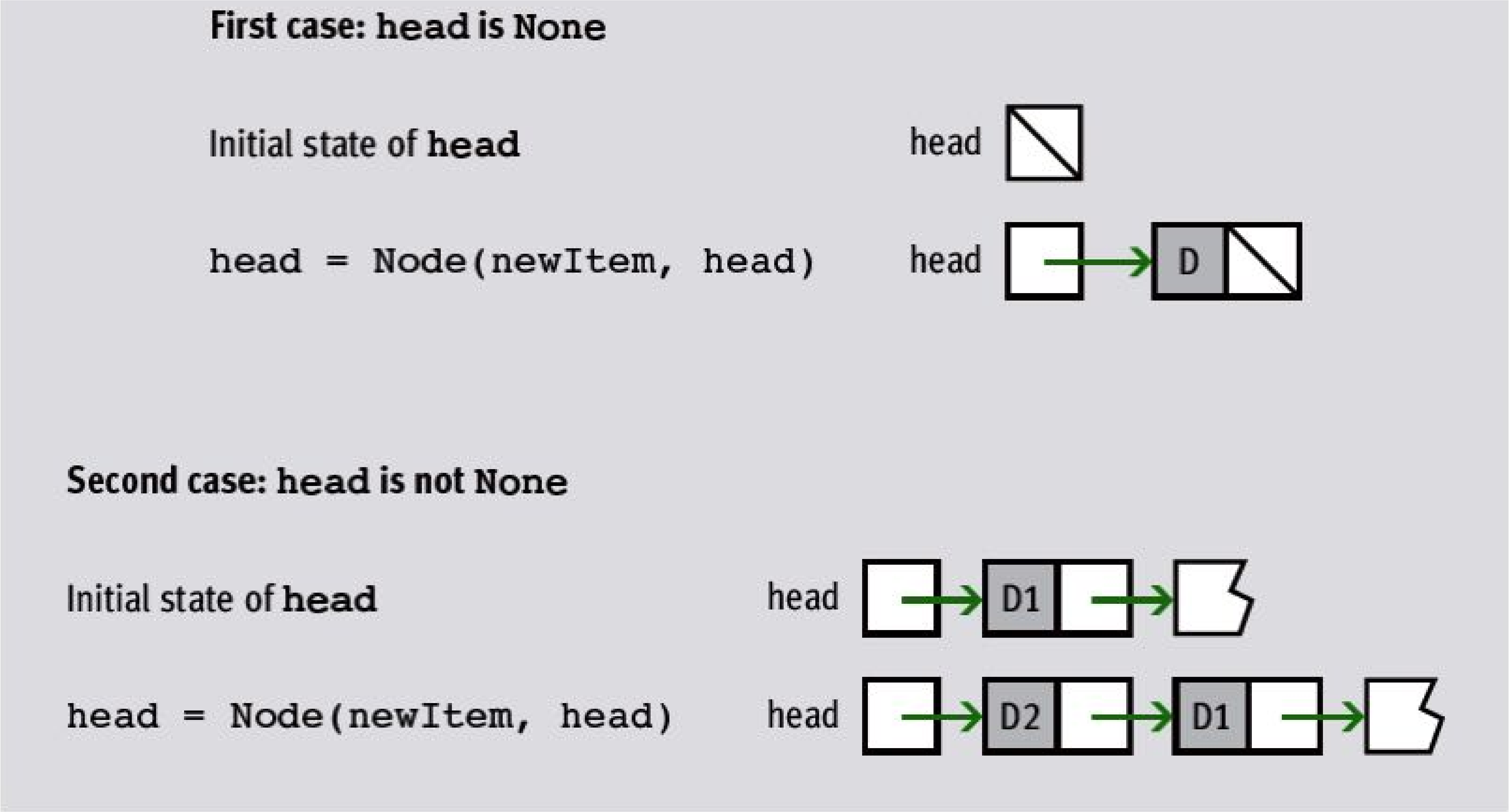
Operations on Singly Linked Structures

- Operations on linked lists are performed by manipulating the links within the structure
  - Emulate index-based operations on a linked structure
- Common operations include:
  - Insertions
  - Traversals
  - Searching
  - Removals
  - etc.

The Singly Linked List ADT

- ADT operations include:
  - linkedList = {head, item1, item2, item3, … itemn}
  - add(item) – adds item to the front of the linked list.
  - append(item) – adds item to the end of the list.
  - insert(pos, item) – adds item to position pos.
  - remove(item) – deletes item from the list.
  - pop() – removes and returns the last item in the list.
  - pop(pos) – removes and returns the item at position pos.
  - index(item) – returns the position of item in the list.
  - search() – removes and returns the last item in the list.
  - is_empty() – removes true if there are no items in the list.
  - size() –returns the number of items in the list.

Inserting at the Beginning



- Uses constant time and memory

Inserting at the Beginning

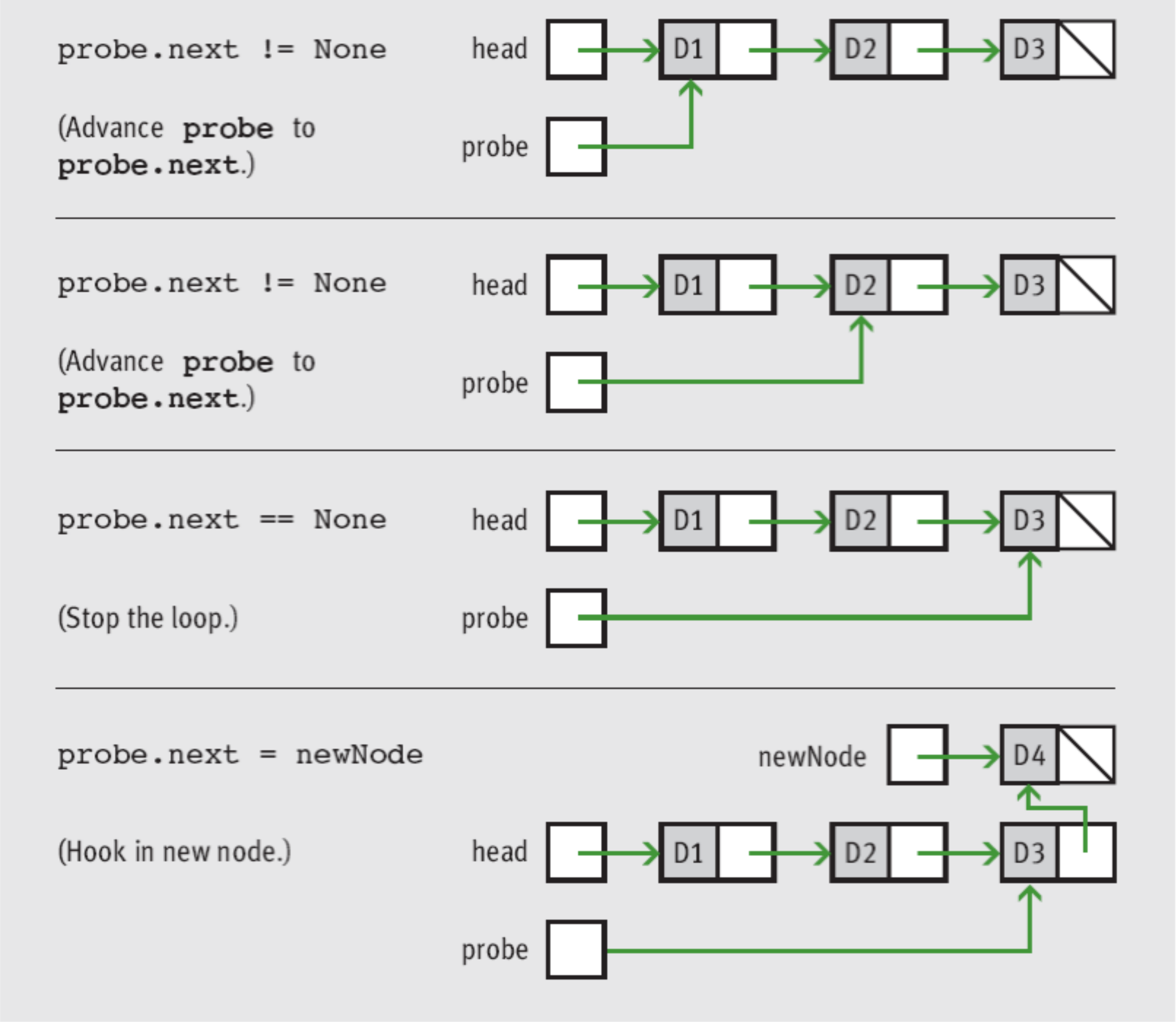
```
# Add item to the front of the linked list
# Handles both an empty list and a list with nodes
    def add(self, data):
        node = Node(data)
        if(node != None):
            node.next = self.head
            self.head = node
```

- Uses constant time and memory

Inserting at the End

- Inserting an item at the end of an array (append in a Python list) requires constant time and memory
  - Unless the array must be resized
- Inserting at the end of a singly linked structure must consider two cases:
  - Empty linked list
  - Linked list is not empty
- Running time: linear in time and constant in memory

Inserting at the End (continued)



Inserting at the End (continued)

- Running time: linear in time and constant in memory

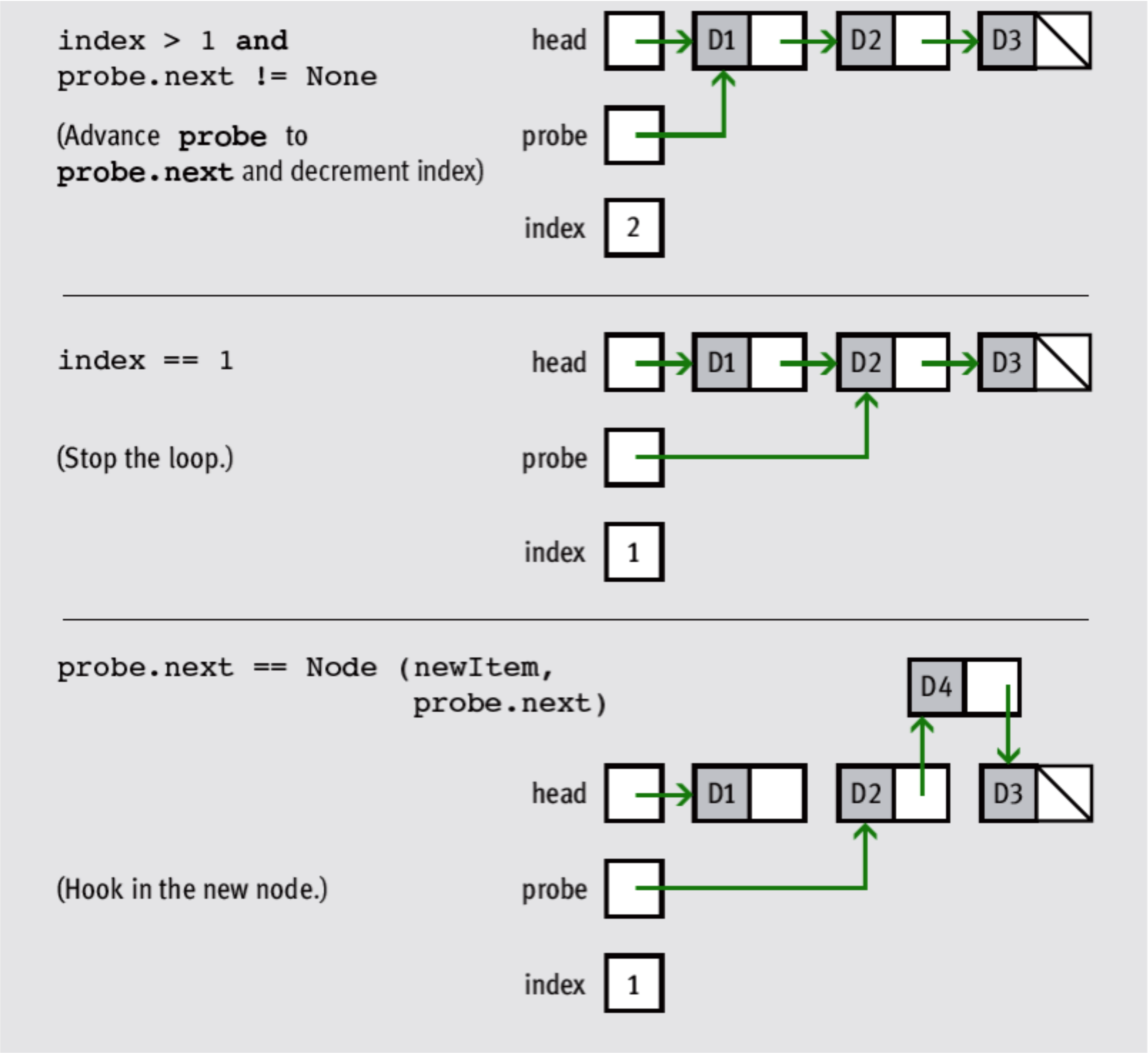
```
#Adds item to the end of the linked list.
# Handles both an empty list and a list with nodes
    def append(self, data):
        newNode = Node(data)
        if(newNode != None):
            if(self.head == None):# list is empty
                self.head = newNode
            else:
                probe = self.head
                while(probe.next != None):#find last node
                    probe = probe.next
                probe.next = newNode
```

Inserting at Any Position

- Insertion at beginning uses code presented earlier
- In other position i, first find the node at position $i - 1$ (if $i < n$) or node at position $n - 1$ (if $i >= n$)

- Linear time performance; constant use of memory

Inserting at Any Position



Linked List class so far...

```
from Node import Node
class LinkedList:
    def __init__(self):
        self.head = None

    # Add item to the front of the linked list
    def add(self, data):
        node = Node(data)
        if(node != None):
            node.next = self.head
            self.head = node

    #adds item to the end of the linked list.
    def append(self, data):
        node = Node(data)
        if(node != None):
            if(self.head == None):
                self.head = node
            else:
                trav = self.head
                while(trav.next != None):#find last node
                    trav = trav.next
                trav.next = node
```

```
#insert(pos, item) – adds item to position pos >= 1.
def insert(self, pos, item):
   size = self.size()
   #print("size pos item", size, pos, item)
   if(pos <= 0 or (size - pos )< -1):
      print("Not enough items in list to insert in that position. Size =",\
               size, "position = ", pos)
      return False
   if(pos == 1):#make new item the first
       newNode = Node(item)
       newNode.next = self.head
       self.head = newNode
   else:#find the position for the new item
        count = 2
        probe = self.head
        while(probe != None and count != pos):
            probe = probe.next
            count += 1
        #Insert after probe
        newNode = Node(item)
        newNode.next = probe.next
        probe.next = newNode
        return True
```

```
#size() –returns the number of items in the list.
def size(self):
    count = 0
    temp = self.head
    while(temp != None):
        count = count + 1
        temp = temp.next
    return count

def printList(self, msg):
    temp = self.head
    print(msg, end = ": ")
    while(temp != None):
        print(temp.get_data(), end=" ")
        temp = temp.next
    print()
```

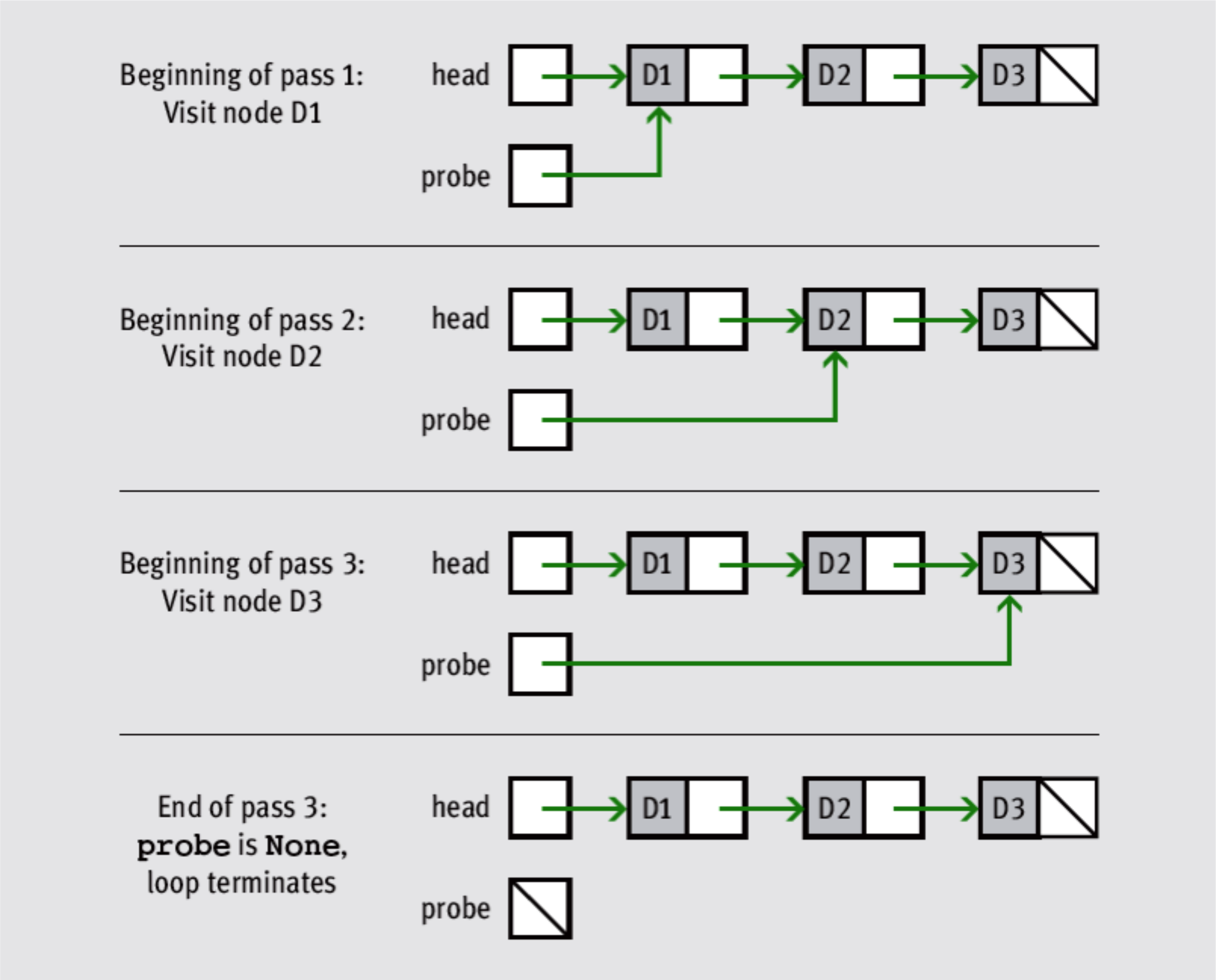
Traversal

- Traversal: Visit each node without deleting it
  - Uses a temporary pointer variable
- Example:

```
probe = head
while probe != None:
    <use or modify probe.data>
    probe = probe.next
```

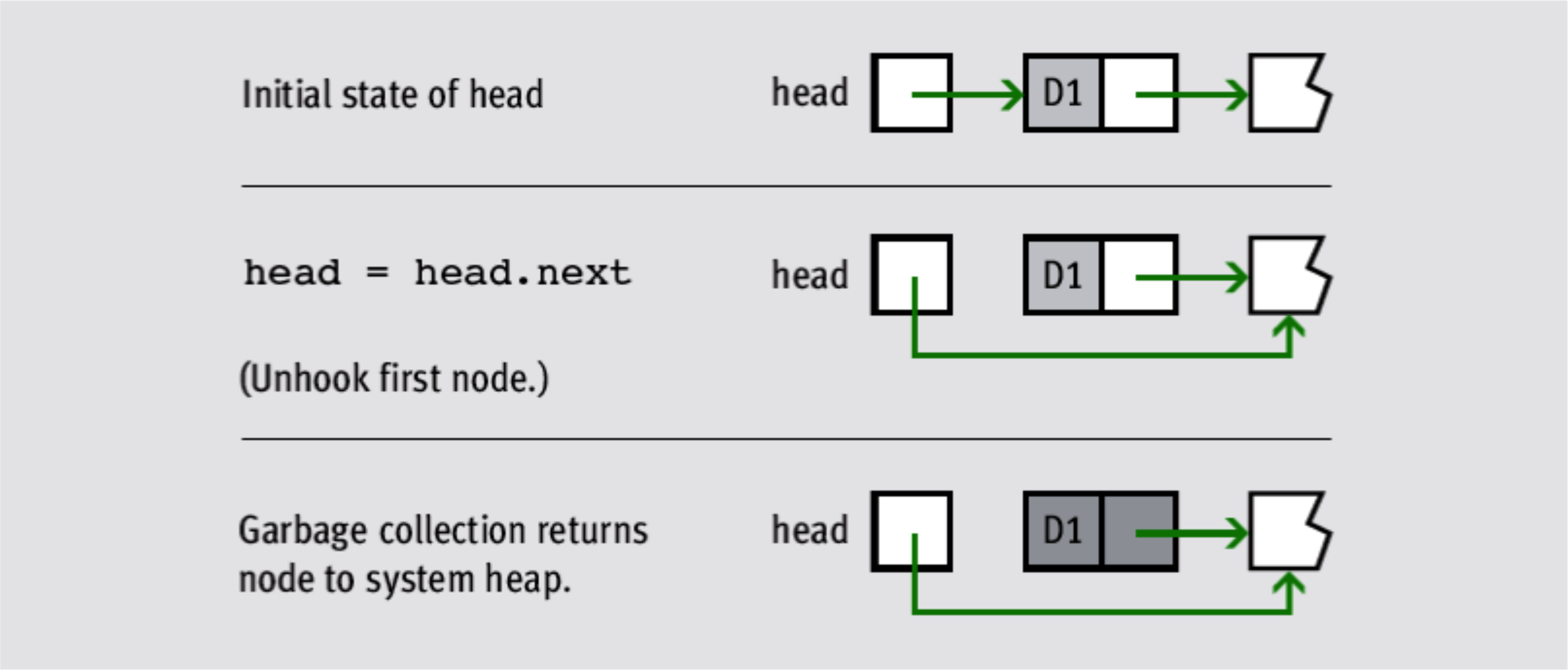
  - None serves as a sentinel that stops the process
- Traversals are linear in time and require no extra memory

Traversal (continued)



Removing at the Beginning

```
# Assumes at least one node in the structure
removedItem = head.data
head = head.next
return removedItem
```

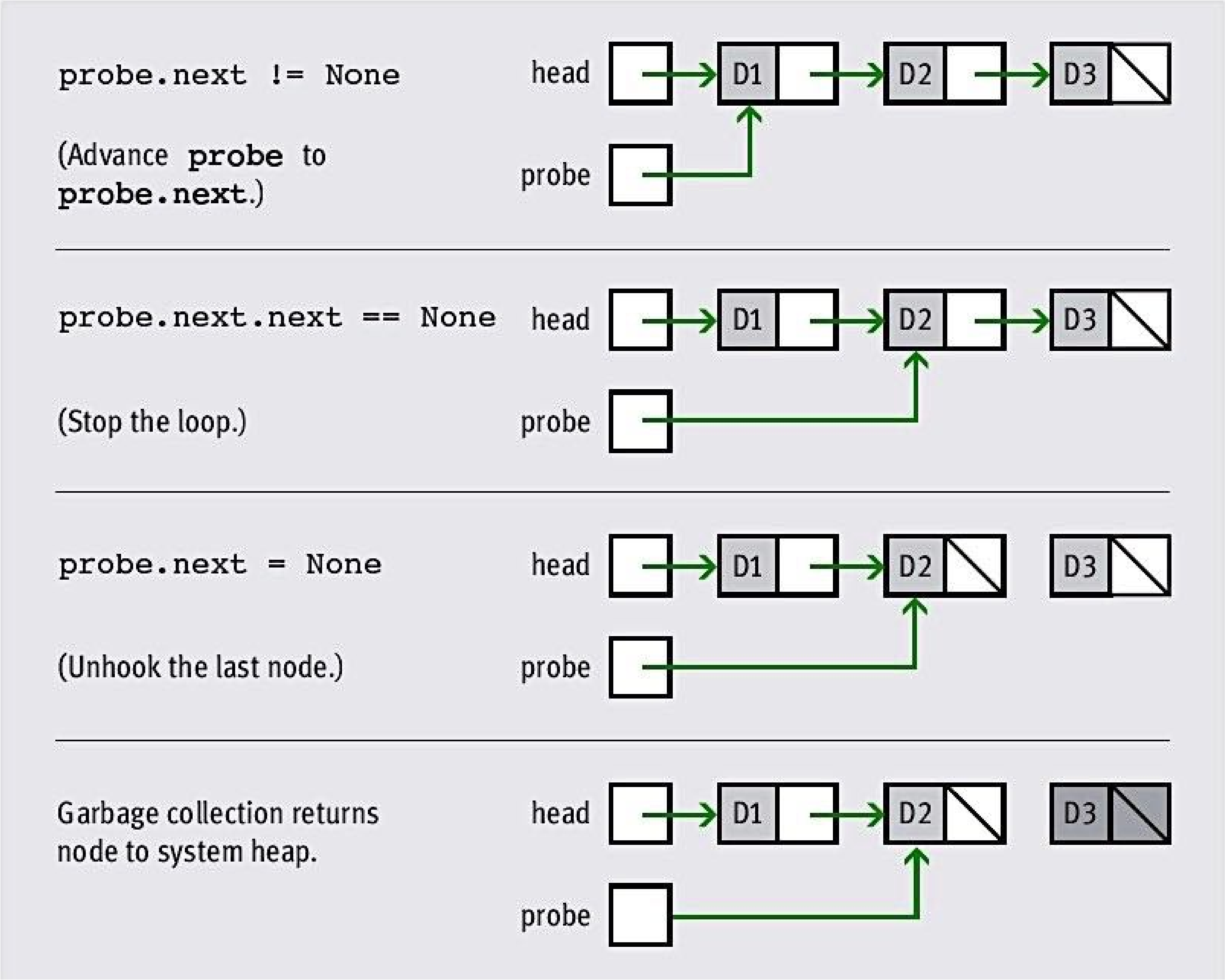


Removing at the End

- Removing an item at the end of an array (pop in a Python list) requires constant time and memory
  - Unless the array must be resized
- Removing at the end of a singly linked structure must consider two cases:

```
# Assumes at least one node in structure
removedItem = head.data
if head.next is None:
    head = None
else:
    probe = head
    while probe.next.next != None:
        probe = probe.next
    removedItem = probe.next.data
    probe.next = None
return removedItem
```

Removing at the End (continued)



Removing at Any Position

- The removal of the ith item from a linked structure has three cases:

```
# Assumes that the linked structure has at least one item
if index <= 0 or head.next is None
    removedItem = head.data
    head = head.next
    return removedItem
else:
    # Search for node at position index - 1 or
    # the next to last position
    probe = head
    while index > 1 and probe.next.next != None:
        probe = probe.next
        index -= 1
    removedItem = probe.next.data
    probe.next = probe.next.next
    return removedItem
```

Replacement

- Replacement operations employ traversal pattern

```
probe = head
while probe != None and targetItem != probe.data:
    probe = probe.next;
if probe != None:
    probe.data = newItem
    return True
else:
    return False
```

- Replacing the ith item assumes 0 <= i < n

```
# Assumes 0 <= index < n
probe = head
while index > 0:
    probe = probe.next
    index -= 1
probe.data = newItem
```

- Both replacement operations are linear on average

Complexity Trade-Off: Time, Space, and Singly Linked Structures

OPERATION	RUNNING TIME
Access at *i*th position	O(n) (average case)
Replacement at *i*th position	O(n) (average case)
Insert at beginning	O(1) (best and worst case)
Remove from beginning	O(1) (best and worst case)
Insert at *i*th position	O(*n*) (average case)
Remove from *i*th position	O(*n*) (average case)

- The main advantage of singly linked structure over array is not time performance but memory performance

Searching

- Resembles a traversal, but two possible sentinels:
  - Empty link
  - Data item that equals the target item
- Example:

```
probe = head
while probe != None and targetItem != probe.data:
    probe = probe.next
if probe != None:
    <targetItem has been found>
else:
    <targetItem is not in the linked structure>
```

- On average, it is linear for singly linked structures

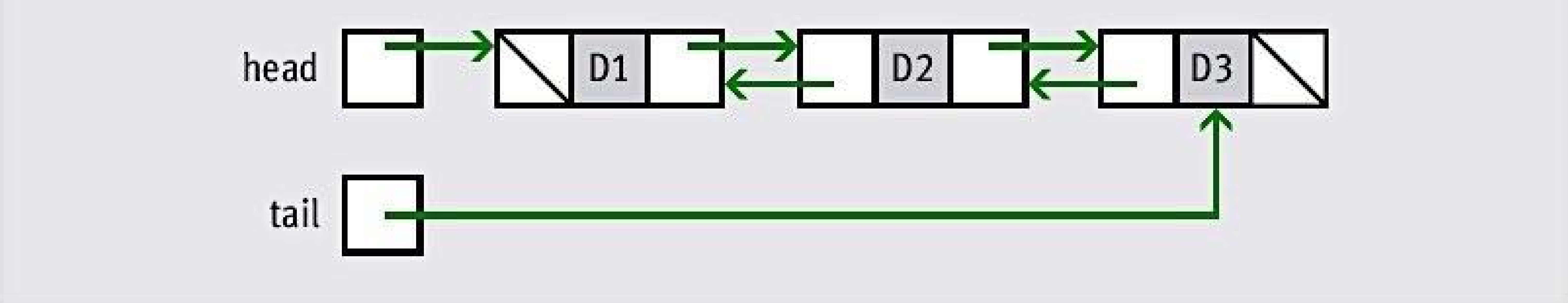
Searching (continued)

- Unfortunately, accessing the ith item of a linked structure is also a sequential search operation
  - We start at the first node and count the number of links until the ith node is reached

```
# Assumes 0 <= index < n
probe = head
while index > 0:
    probe = probe.next
    index -= 1
return probe.data
```

- Linked structures do not support random access
  - Can't use a binary search on a singly linked structure
  - Solution: Use other types of linked structures

Doubly Linked Structures

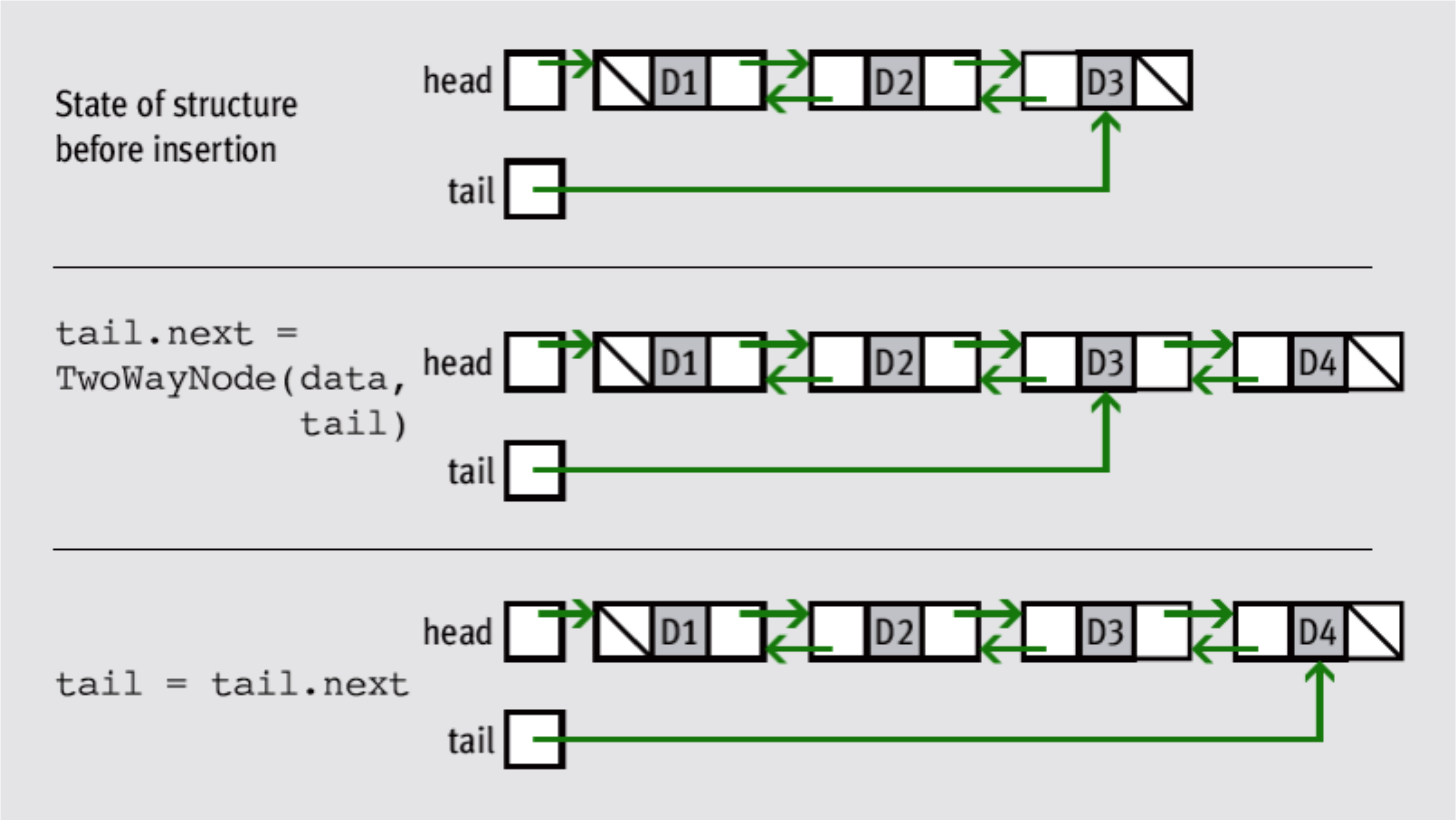



```
class Node(object):

    def __init__(self, data, next = None):
        """Instantiates a Node with default next of None"""
        self.data = data
        self.next = next

class TwoWayNode(Node):

    def __init__(self, data, previous = None, next = None):
        """Instantiates a TwoWayNode."""
        Node.__init__(self, data, next)
        self.previous = previous
```

Doubly Linked Structures (continued)

```
"""
File: testtwowaynode.py
Tests the TwoWayNode class.
"""

from node import TwoWayNode

# Create a doubly linked structure with one node
head = TwoWayNode(1)
tail = head

# Add four nodes to the end of the doubly linked structure
for data in xrange(2, 6):
    tail.next = TwoWayNode(data, tail)
    tail = tail.next

# Print the contents of the linked structure in reverse order
probe = tail
while probe != None:
    print probe.data
    probe = probe.previous
```

Doubly Linked Structures (continued)



Summary

- Collections are objects that hold 0+ other objects
  - Main categories: Linear, hierarchical, graph, and unordered
  - Collections are iterable
  - Collections are thus abstract data types
- A data structure is an object used to represent the data contained in a collection
- The array is a data structure that supports random access, in constant time, to an item by position
  - Can be two-dimensional (grid)

Summary (continued)

- A linked structure is a data structure that consists of 0+ nodes
  - A singly linked structure's nodes contain a data item and a link to the next node
  - Insertions or removals in linked structures require no shifting of data elements
  - Using a header node can simplify some of the operations, such as adding or removing items

Questions?

@rschifan

schifane@di.unito.it

http://www.di.unito.it/~schifane