SINGAPORE INSTITUTE OF TECHNOLOGY

ICT1008 Data Structures and Algorithms

Lecture 1: Basic Data Structures

Agenda

- Abstract Data Types
  - Arrays
  - Stacks
  - Queues
  - Linked Lists

Recommended reading

- Algorithms by Robert Sedgewick and Kevin Wayne. Addison-Wesley Professional. 4th edition, 2011.
  – Chapter 1.3
- Runestone Interactive book: "Problem Solving with Algorithms and Data Structures Using Python"
  – Section "Basic Data Structures"

Review

Data Structures

- Components of the data
- Organization of the data

Algorithms

- How to access data
- How to sort/search data
- Available operations (methods) to operate on data

Abstract Data Types (ADT)

- A Data Type is
  - A set of values
  - A set of operations on those values
- An Abstract Data Type (ADT) is a data type whose representation/implementation is hidden from the users.
- Difference between ADT and Primitive Data Type?

ADT - example

```
class Counter:

    def __init__(self,id):...

    def incrementByOne(self):...
```

Why ADT?

ADT is important

- supports **encapsulation** in program design
- specifies precisely the problem
- describes algorithms and data structures (as API) to be used by clients

Using ADT

When using an ADT:

we focus on the operations.

```
class Counter:

    def __init__(self,id):...

    def incrementByOne(self):...
```

```
aCounter = Counter("studentCounter")
aCounter.incrementByOne()
```

We do not need to know how a data type is implemented in order to know how to use it.

ADT and Python

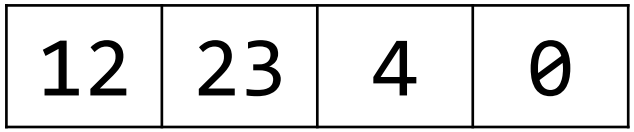
In this course, we will use Python to specify an ADT as well as to implement the ADT

Python Dev Tools

- Online: Colab
- WSL, python on windows

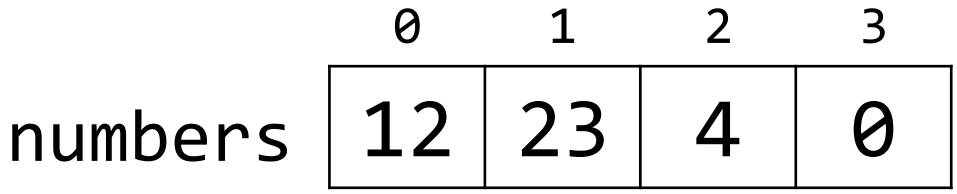
Array

A linear data structure consisting of a fixed number of data items of the same type.



An array of 4 integers

Array operations




```
numbers[0] == 12
numbers[1] == 23
```

Access array element directly through index

Array Implementation in Python

```
numbers = [12, 13, 4, 0]
numbers[2] = 100
print ("numbers ", numbers)
```

```
numbers  [12, 13, 100, 0]
```

Array Implementation in Python

```
 8  import random
 9  SIZE = 10
10  #Create an array of 10 and
11  #initialize all elements to 0
12  numbers = SIZE*[0]
13  for i in range(SIZE):
14      numbers[i] = random.randrange(100, 1000)
15  print 'numbers: ',numbers
```

```
numbers:  [705, 500, 791, 382, 336, 344, 424, 913, 375]
```

Multi-dimensional array

matrix

	0	1	2	3
0	4	6	7	2
1	3	5	1131	123
2	45	6	33	323

```
matrix[0][0] == 4
matrix[1][2] == 1131
```

Array can be `multi-dimensional`

- Array of arrays e.g. [[], []. … []]
- `More than one index` can be used to access elements in a particular position

Multi-dimensional array in Python

```
X_SIZE = 3
Y_SIZE = 4
# Creates a list containing X_SIZE sub-lists
# Each sub-list contains Y_SIZE elements initialized to 0
matrix = [[0 for x in range(Y_SIZE)] for x in range(X_SIZE)]
for i in range(X_SIZE):
    for j in range(Y_SIZE):
        matrix[i][j] = (random.randint(100, 1000))
    print matrix[i]
```

Output?

```
[431, 464, 806, 569]
[718, 329, 650, 796]
[425, 117, 373, 464]
```

Stack

- Stack is a linear data structure which holds multiple elements of a single data type.
- Rule: **Last-In-First-Out** (LIFO or FILO)

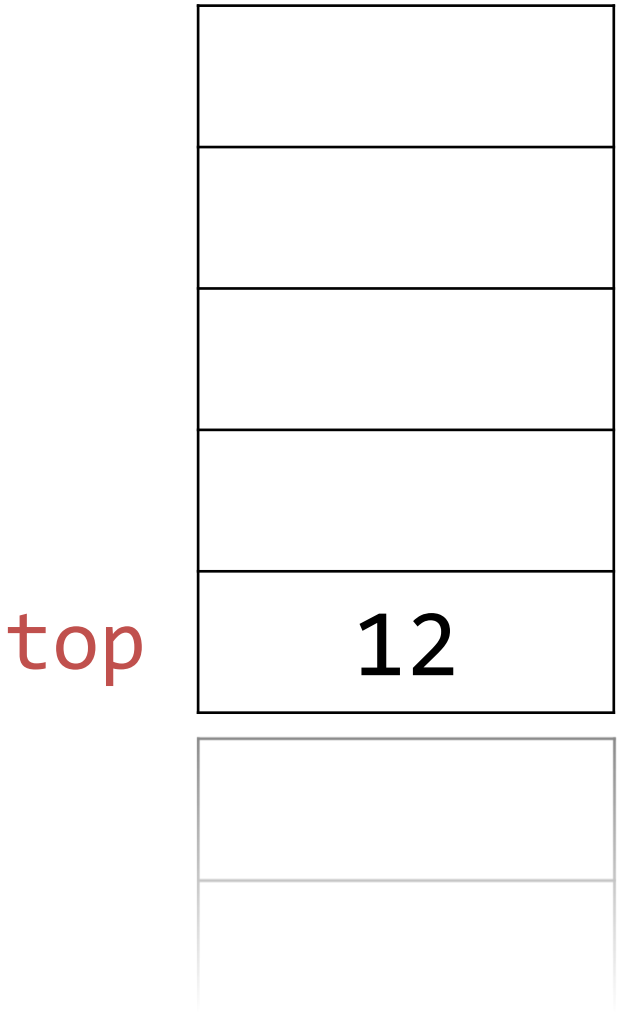
Stack Operations

- `push(value)`
  - Add `value` to the top of the stack
- `pop()`
  - Remove and return the value on top of the stack

Stack operations

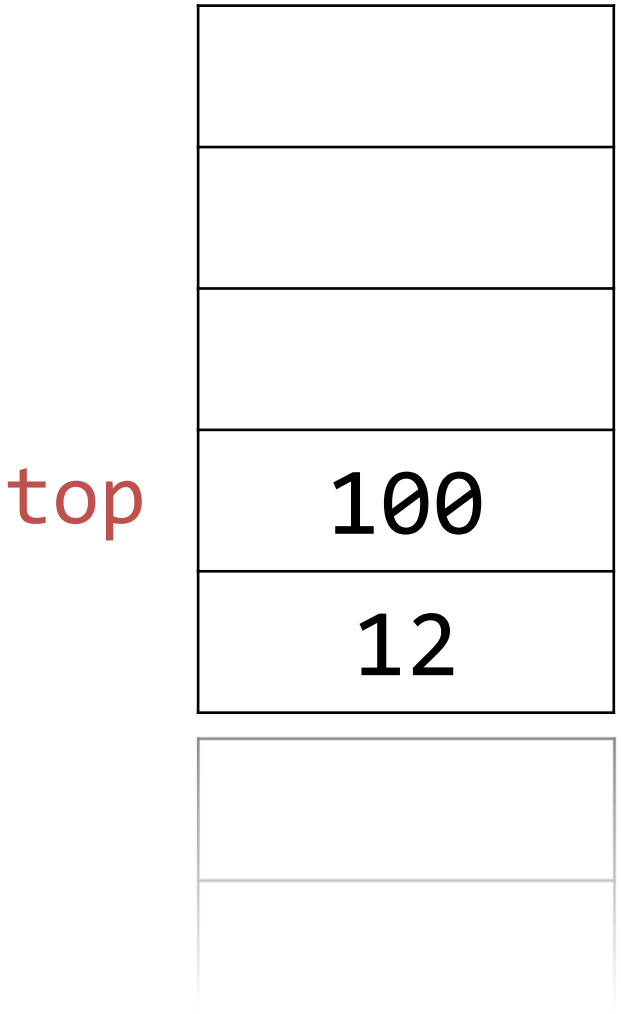

```
push(12)
```

Stack operations



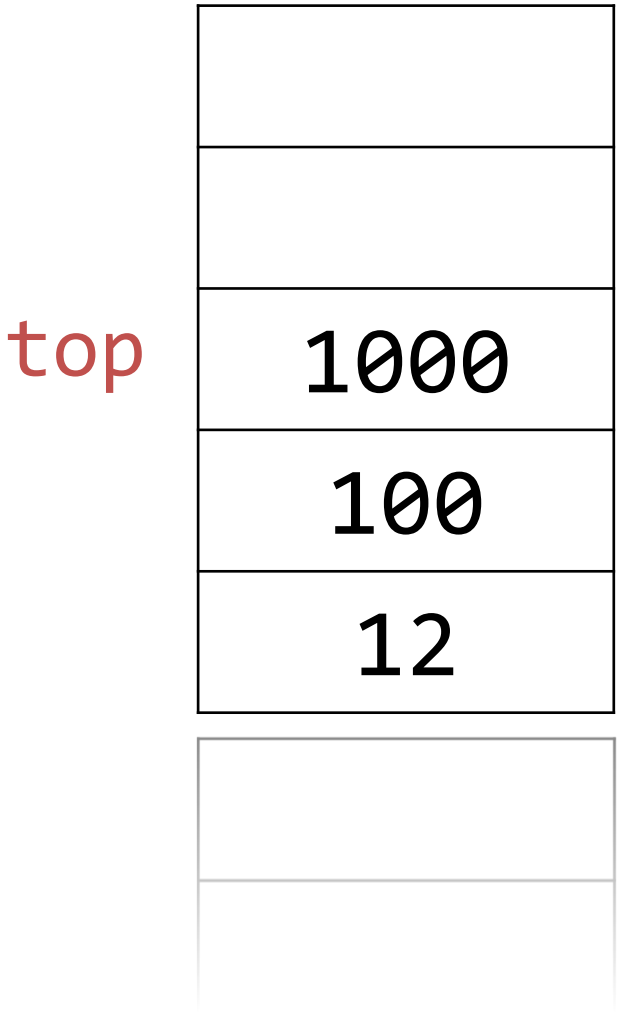

```
push(100)
```

Stack operations



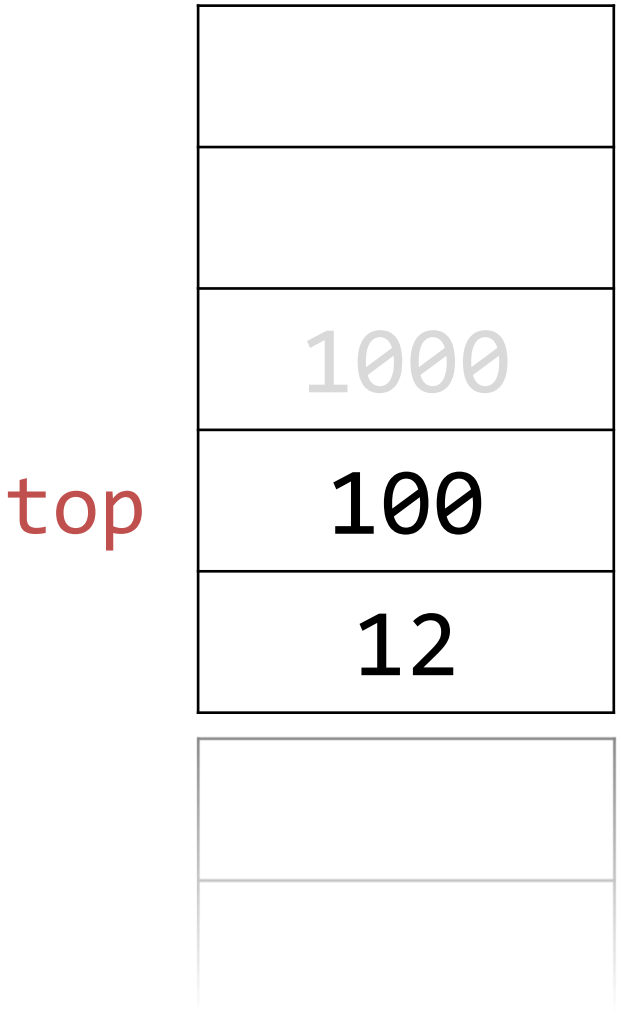

```
push(1000)
```

Stack operations



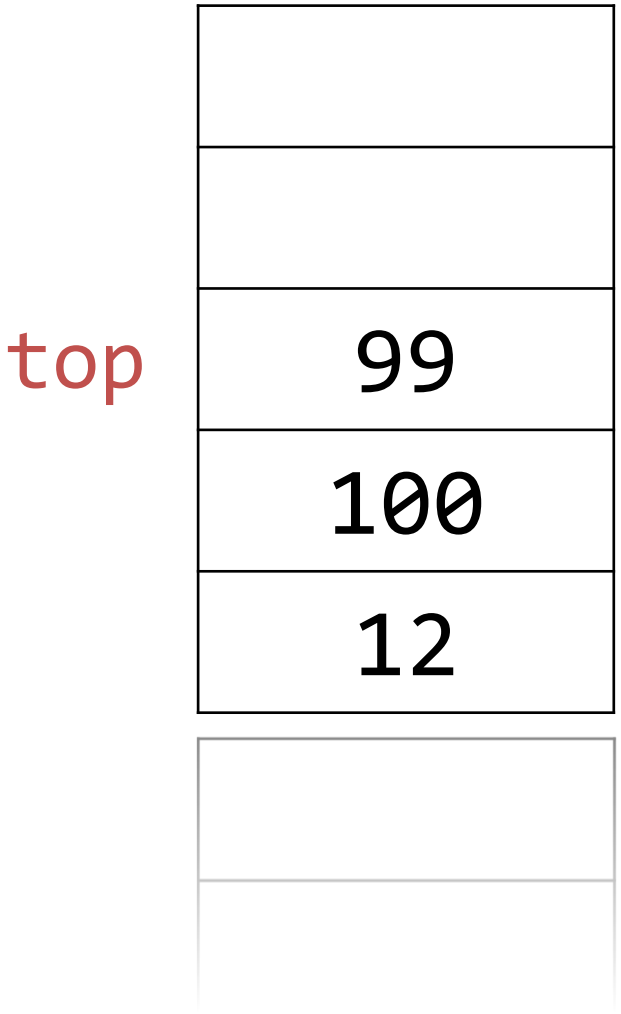
pop()

Stack operations




```
return 1000
push(99)
```

Stack operations




```
return 1000
push(99)
```

Stack ADT

- Data
  - `top`: to keep track of the top index
  - `data`: some linear structure to store data
- Operations
  - `push(value)`
  - `pop()`
  - …

Stack – Python implementation

- Many ADT (including stack, queue…) can be implemented easily using Python built-in functions.
- However, *to understand the algorithms*, we try not to use built-in functions here.

Stack – Python implementation

```
class Stack:
    def __init__(self):
        self.top = -1
        self.data = []

    def push(self, value):...

    def pop(self):...
```

init() initializes the top index to -1 and empty data

```
aStack = Stack()
aStack.push(10)
aStack.push(100)
aStack.pop()
aStack.push(99)
```

Output?

```
Push  10
[10]
Push  100
[10, 100]
Pop the top value
[10]
Push  99
[10, 99]
```

Stack operations - Algorithms

`push(value)`

1. extend the size of the stack by 1
2. increase the top index by one
3. assign `value` to the element at the top

`pop()`

1. read the `value` of the element at the top
2. delete the element at the top
3. decrease the top index by 1
4. return the `value`

Stack – Python implementation

```
def push(self, value):
    #increment the size of data using append()
    self.data.append(0)
    self.top += 1
    self.data[self.top] = value

def pop(self):
    value = self.data[self.top]
    #delete the top value using del
    del self.data[self.top]
    self.top -= 1
    return value
```

What if the stack is empty?
Calling `pop()` will cause an exception
`IndexError: list index out of range`

Stack – Other operations

- `isEmpty()`
  - return `true` if the stack is empty; `false` otherwise
- `peek()`
  - return the value at the top without removing the value from the stack
- …

Queue

- Queue is a linear data structure which holds multiple elements of a single data type.
- Rule: **First-In-First-Out**
  - adding at the rear
  - removing at the front

Queue Operations

- `enqueue(value)`
  - Add `value` to the rear of the queue
- `dequeue()`
  - Remove and return the value of the item from the front of the queue

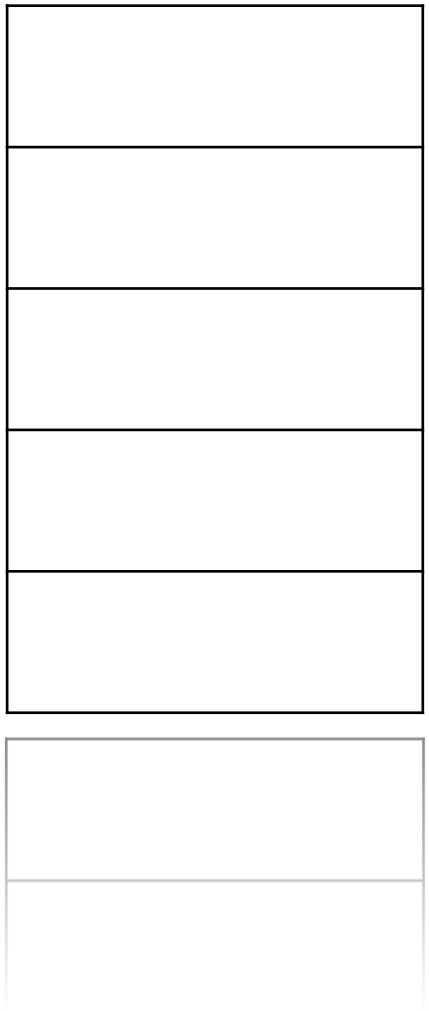
Queue ADT

- Data
  - `rear`: keep track of the rear index
  - `data`: some linear structure to store all the elements in the queue
  - #assumption: front is always at index 0
- Operations
  - `enqueue(value)`
  - `dequeue()`
  - …

Queue ADT

- What to do when initializing a queue?
  - `rear` = -1
  - `data`: empty

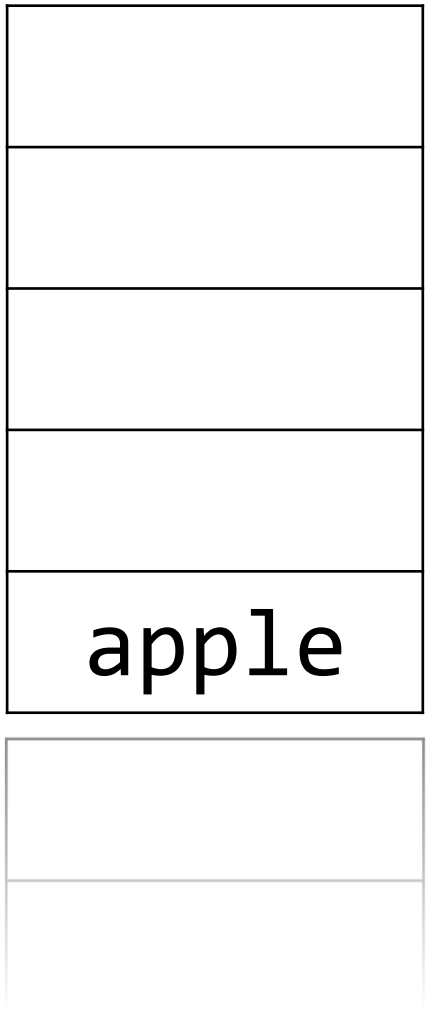
Queue operations



```
enqueue("apple")
```

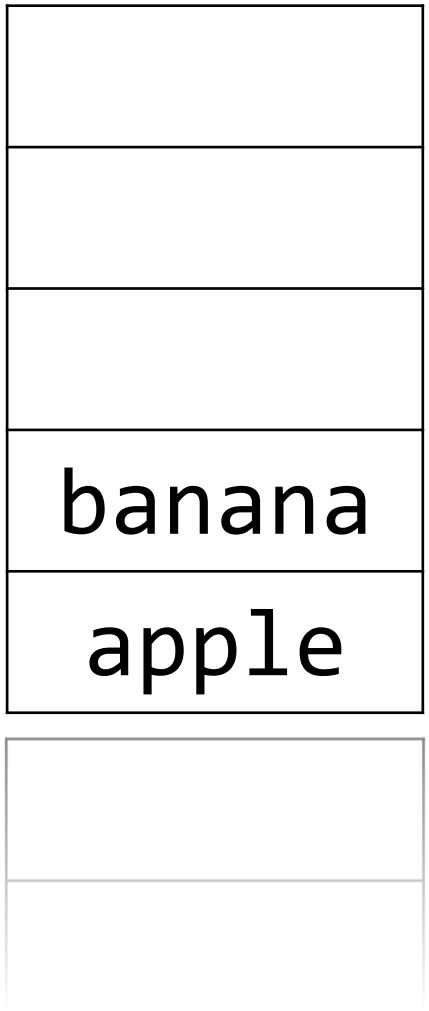
rear

Queue operations



enqueue(“banana”)

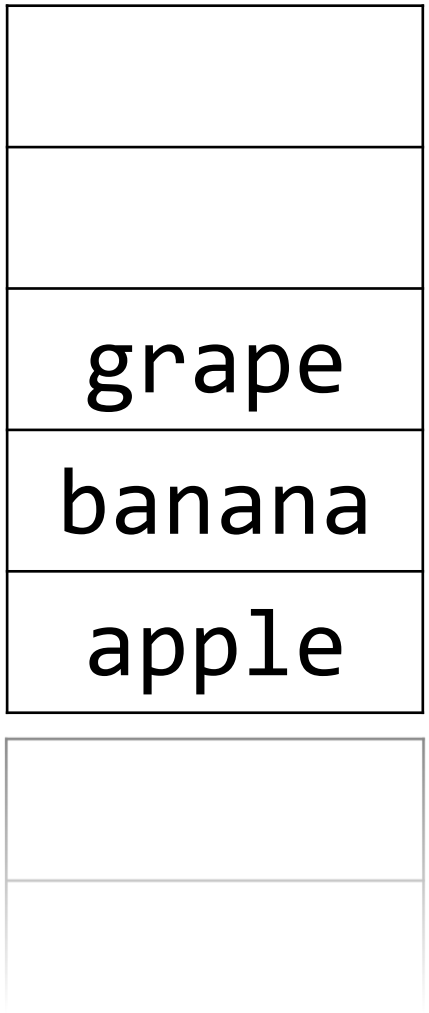
Queue operations



enqueue("grape")

rear

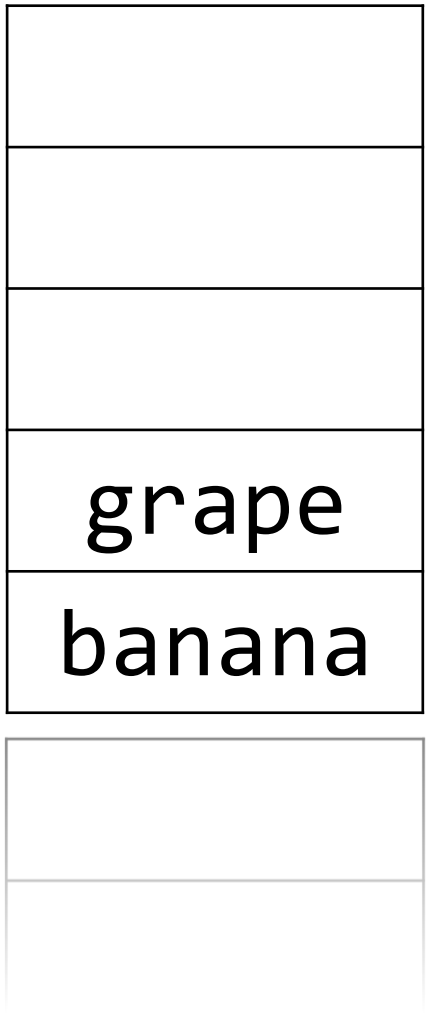
Queue operations



dequeue()

rear

Queue operations




```
dequeue()
return: “apple”
```

Queue ADT - Python

```
class Queue:
    def __init__(self):
        self.rear = -1
        self.data = []

    def enqueue(self, value):...
    def dequeue(self):...
```

```
aQueue = Queue()
aQueue.enqueue("apple")
aQueue.enqueue("pear")
aQueue.enqueue("grape")
aQueue.dequeue()
aQueue.enqueue("banana")
aQueue.dequeue()
```

```
Enqueue  apple
['apple']
Enqueue  pear
['apple', 'pear']
Enqueue  grape
['apple', 'pear', 'grape']
Dequeue the first value
['pear', 'grape']
Enqueue  banana
['pear', 'grape', 'banana']
Dequeue the first value
['grape', 'banana']
```

Queue Operations - Algorithms

`enqueue(value)`

1. extend the size of the queue by 1
2. increase the rear index by one
3. assign `value` to the element at the rear index

`dequeue()`

1. read the `value` at index `0` (front)
2. delete the element at index `0`
3. decrease the rear index by one
4. return `value`
5. Exception
   - If the queue is empty: print an error message

Queue – Python implementation

```
def enqueue(self, value):
    #append value to the end of data
    self.data.append(value)
    self.rear += 1

def dequeue(self):
    value = self.data[0]
    #delete the value at index 0 using del
    del self.data[0]
    self.rear -= 1
    return  value
```

Exception

- If the queue is empty: print an error message

Singly Linked-list

A singly linked list is a `linear` data structure in which each element (node) consists of two items:

1. Data.
2. A reference (`link`) to the next node in the list.

SinglyListNode ADT

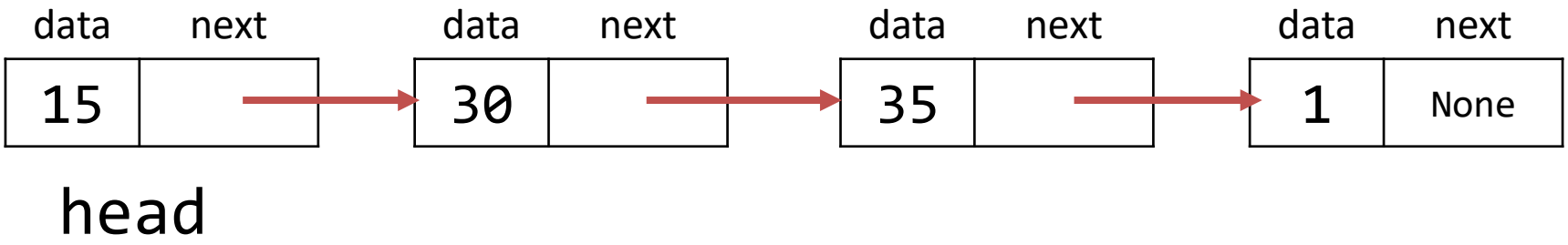
Each node consists of two items

1. Data
2. Link to the next node.

```
class SinglyListNode:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Initialize a new node with `data` and `None` link to the next node.
The link can be set later.
Note: `None` is similar to `null` pointer.

Singly linked list visualization



How to access elements in this list?

- Start from head node
- Use next pointer to access next node in the list

Singly linked list operations

- `insertAtHead(node)`
  - insert node at the beginning of the list
- `search(value)`
  - search and return the node whose data is equal to `value`
- `delete(value)`
  - delete the node whose data is equal to `value`

Singly linked list ADT

- Data
  - head: the head node of the list
- Operations
  - `insertAtHead(node)`
  - `search(value)`
  - `delete(value)`

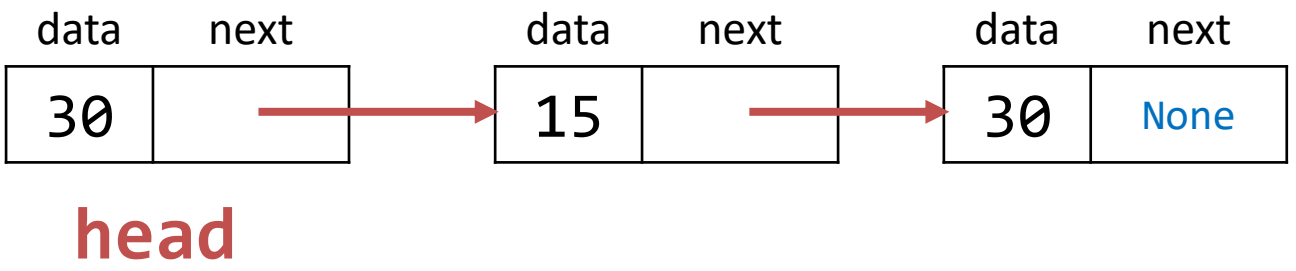
Singly linked list ADT in Python

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def search(self, value):...
    def insertAtHead(self, node):...
    def delete(self, value):...
```

Insert algorithm

`insertAtHead(node)`

1. if the list is empty (head is None) then
   assign node to head
2. else `node.next = head`
   `head = node`



Insert algorithm

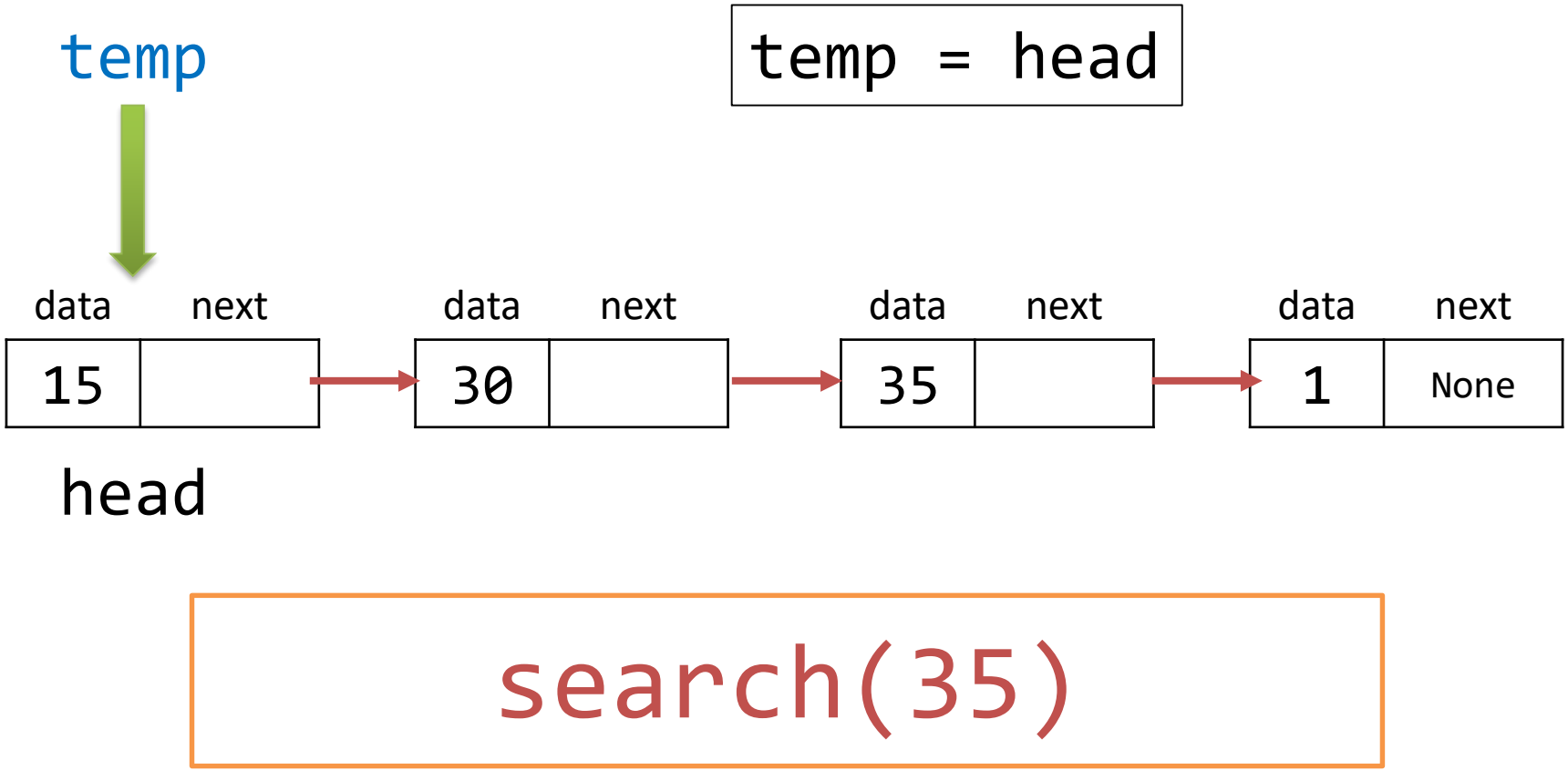
```
def insertAtHead(self, node):
    if self.head is None:
        self.head = node
    else:
        node.next = self.head
        self.head = node
```

Search algorithm

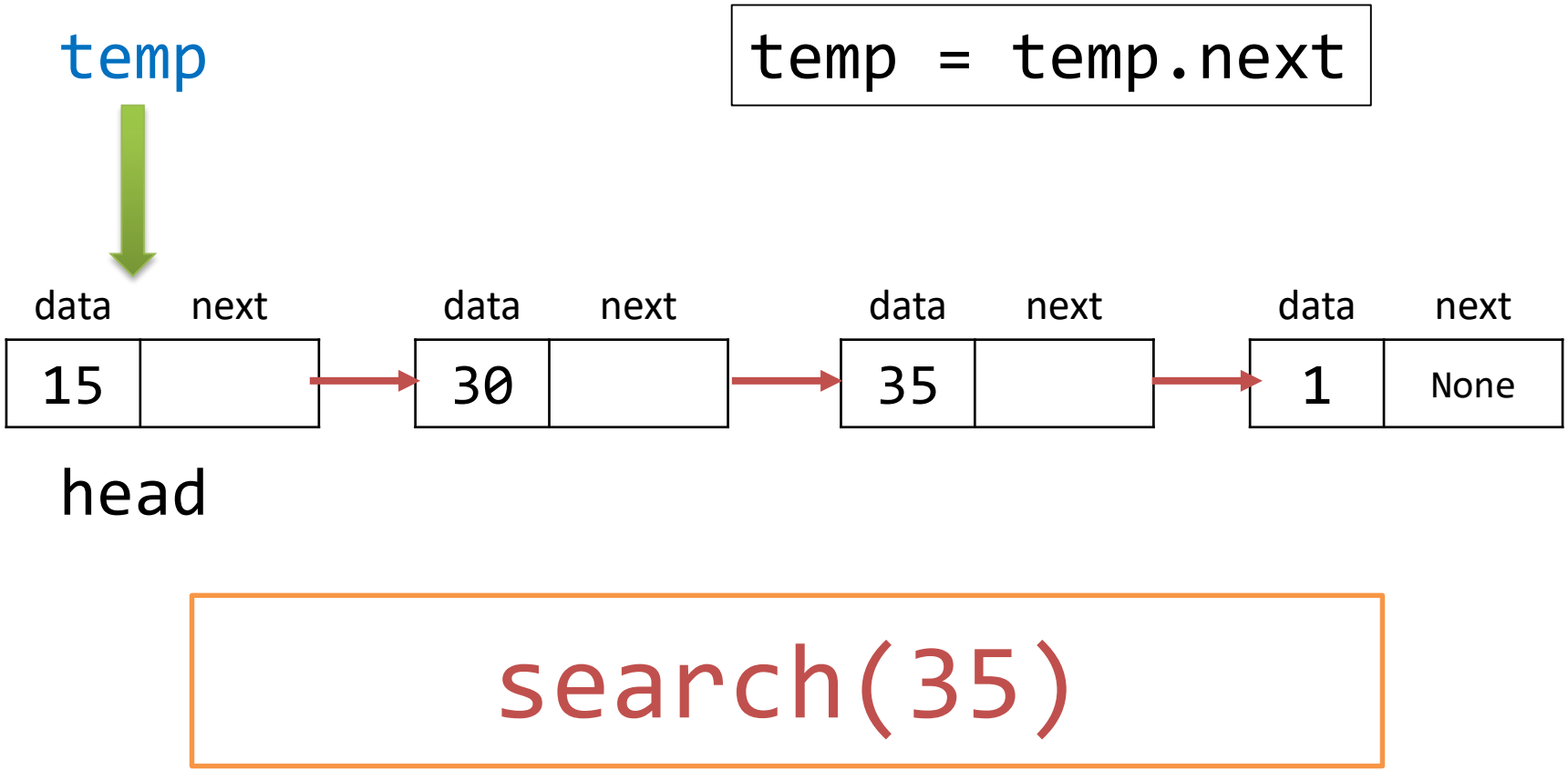
`search(value)`

1. Start from the `head` node
2. Compare the `data` at the node with the `value`
3. if the `data` is not the same as `value`
   - » Go to the next node
4. else
   - » Return the node
5. Repeat 2, 3 or 4 till end of the list

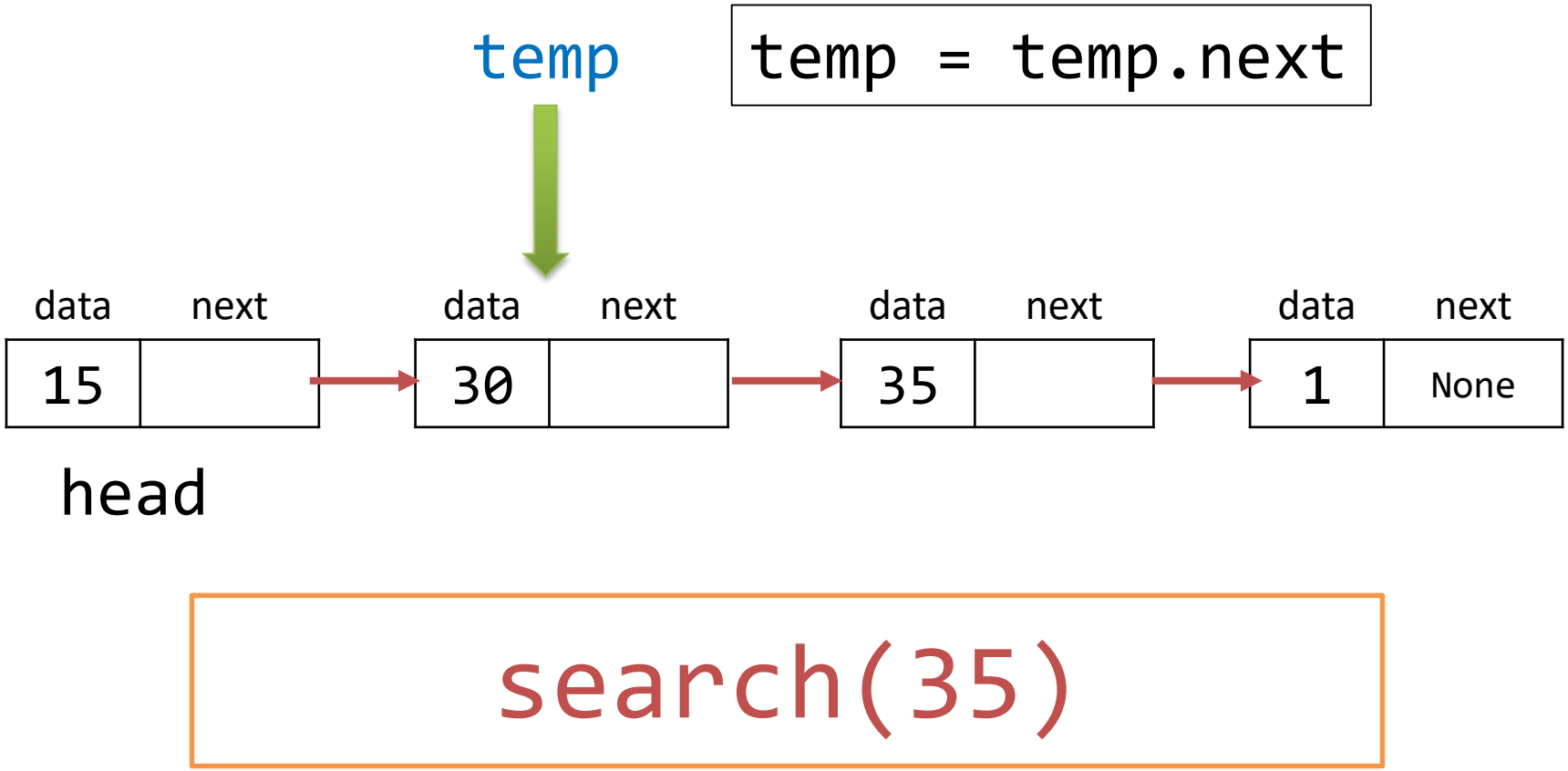
Search algorithm



Search algorithm



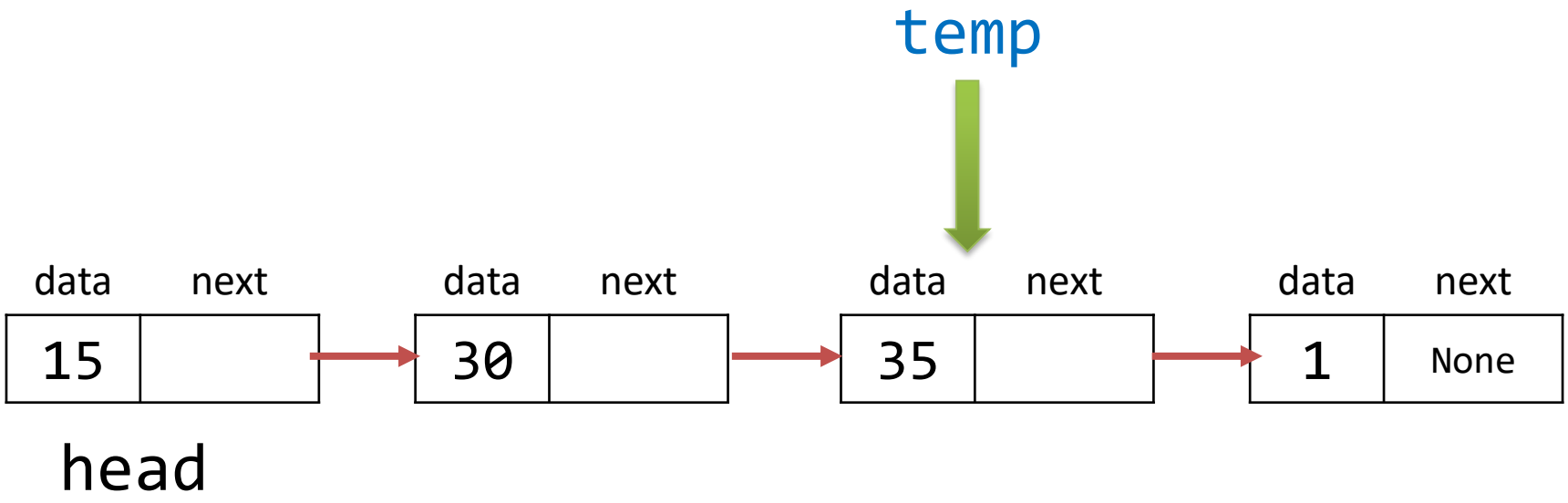
Search algorithm




```
search(35)
```

Search algorithm

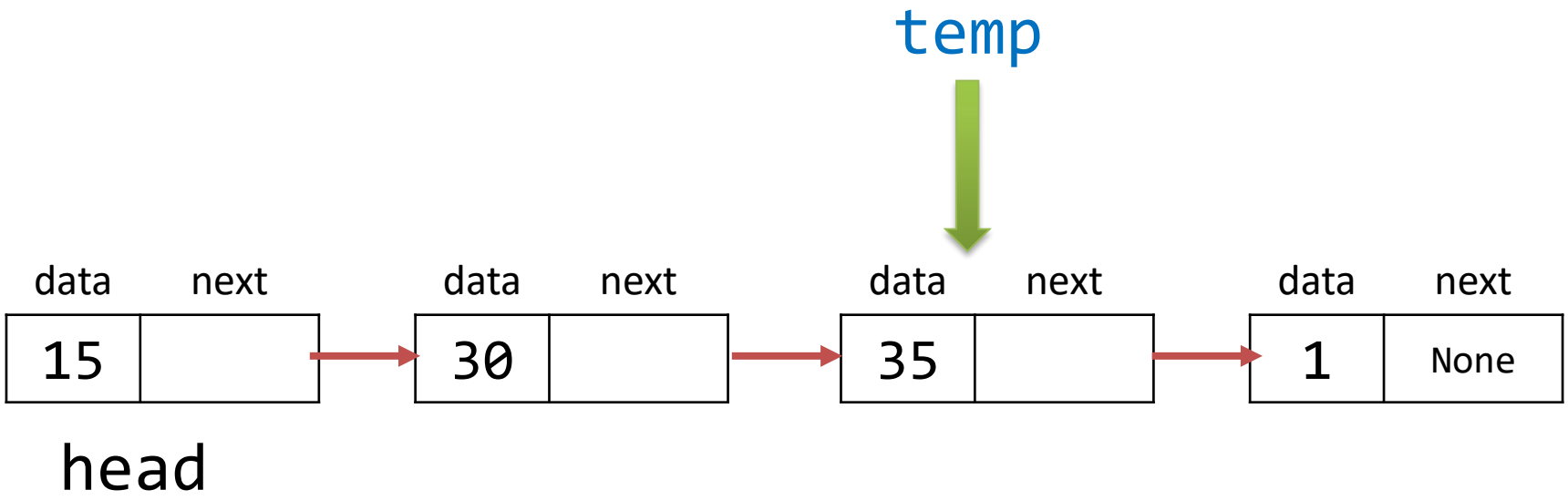
```
temp = temp.next
```




```
search(35)
```

Search algorithm

```
return temp
```



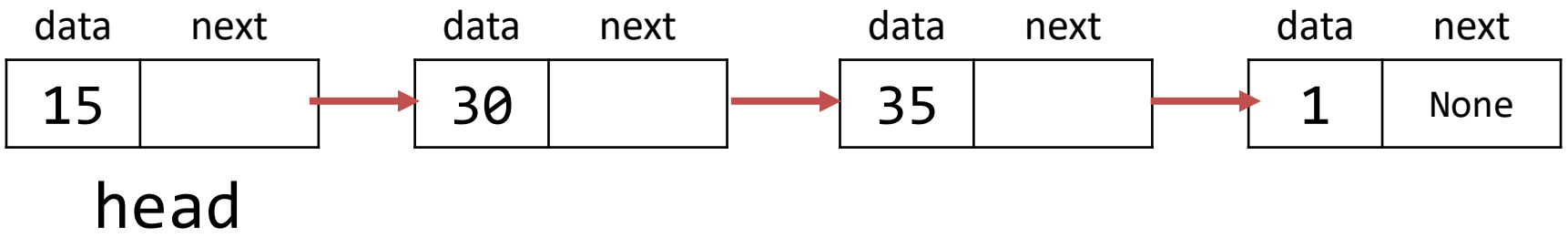

```
search(35)
```

Search algorithm in Python

```
#return the value of the node at index
def search(self, value):
    temp=self.head
    while temp is not None:
        if temp.data is value:
            return temp
        temp = temp.next
    print 'Search Error: Value not found'
```

Delete algorithm

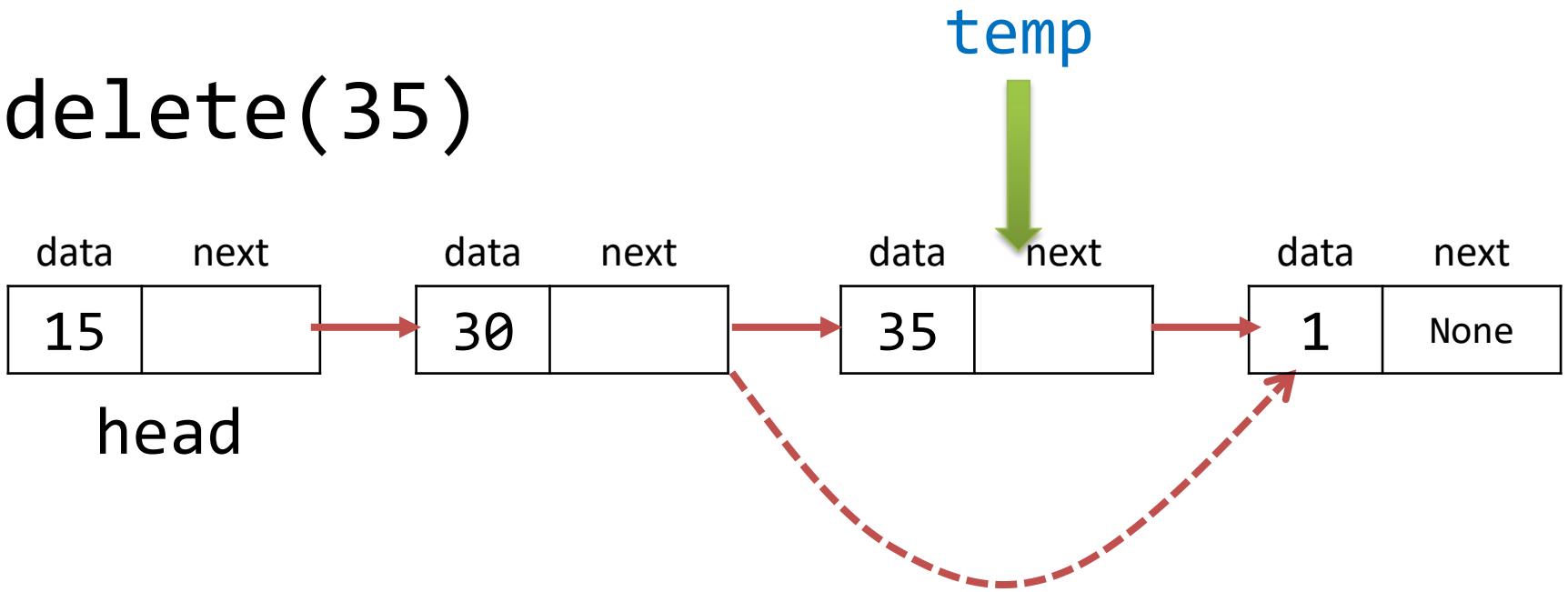
```
delete(35)
```




```
delete(value)
```

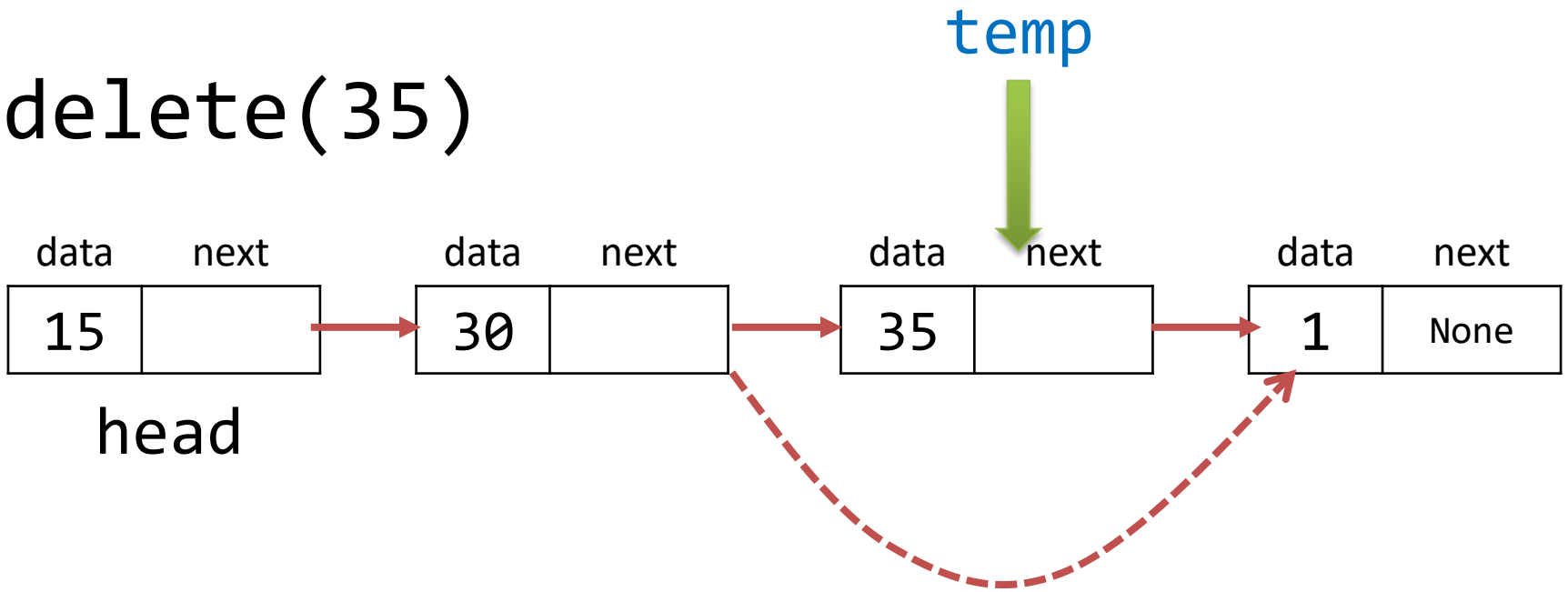
1. Start at the `head` node
2. Search for the node whose data = `value`
3. ?

Delete algorithm



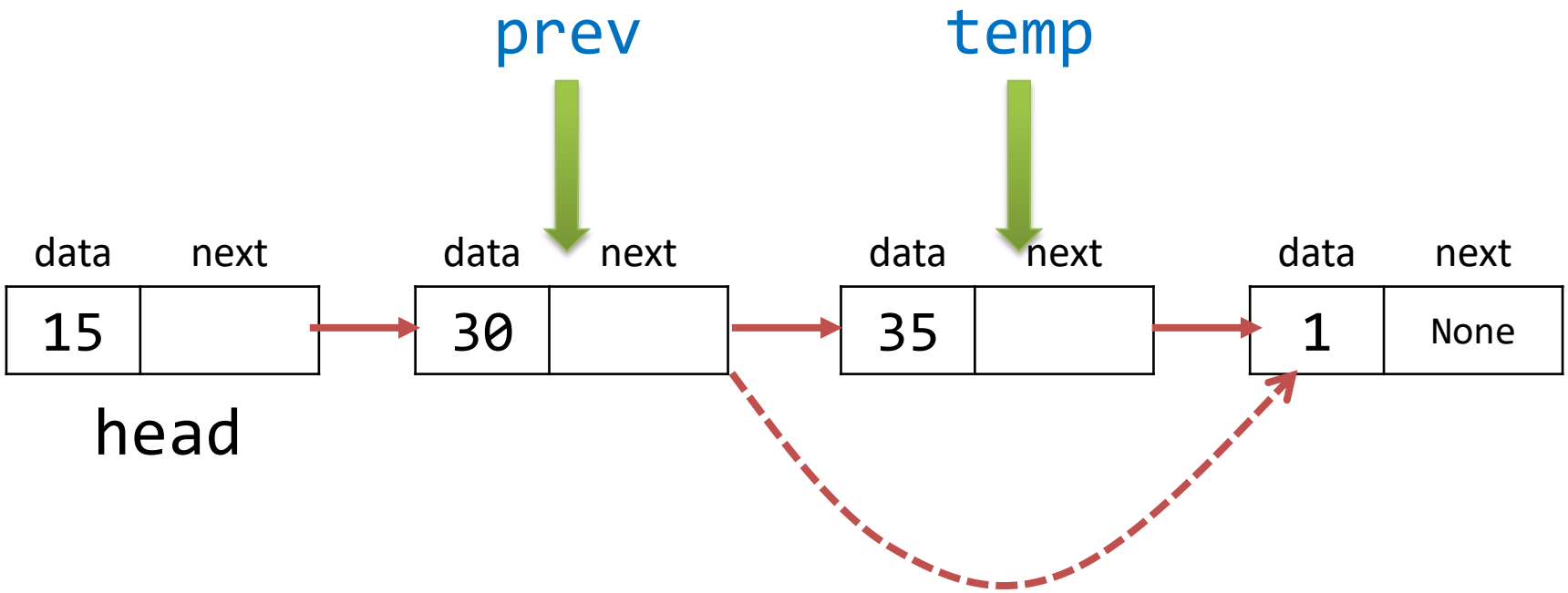
Assuming that we are at node whose data = 35.
How do we delete the node at `temp`?

Delete algorithm



We could not delete the node at `temp`
if we do not know the node before `temp`.

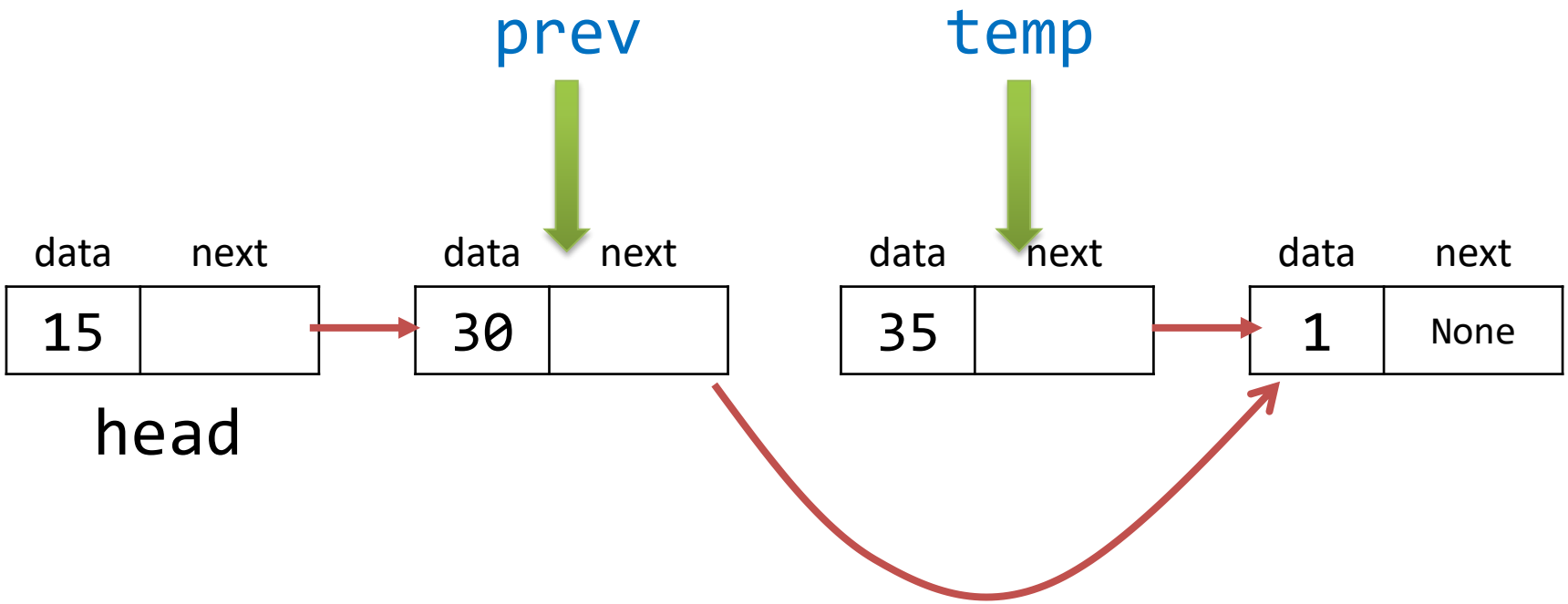
Delete algorithm




```
delete(value)
```

1. Start at the head node
2. Search for the node whose data = `value` and keep track of the `previous` node of temp
3. `prev.next = temp.next`

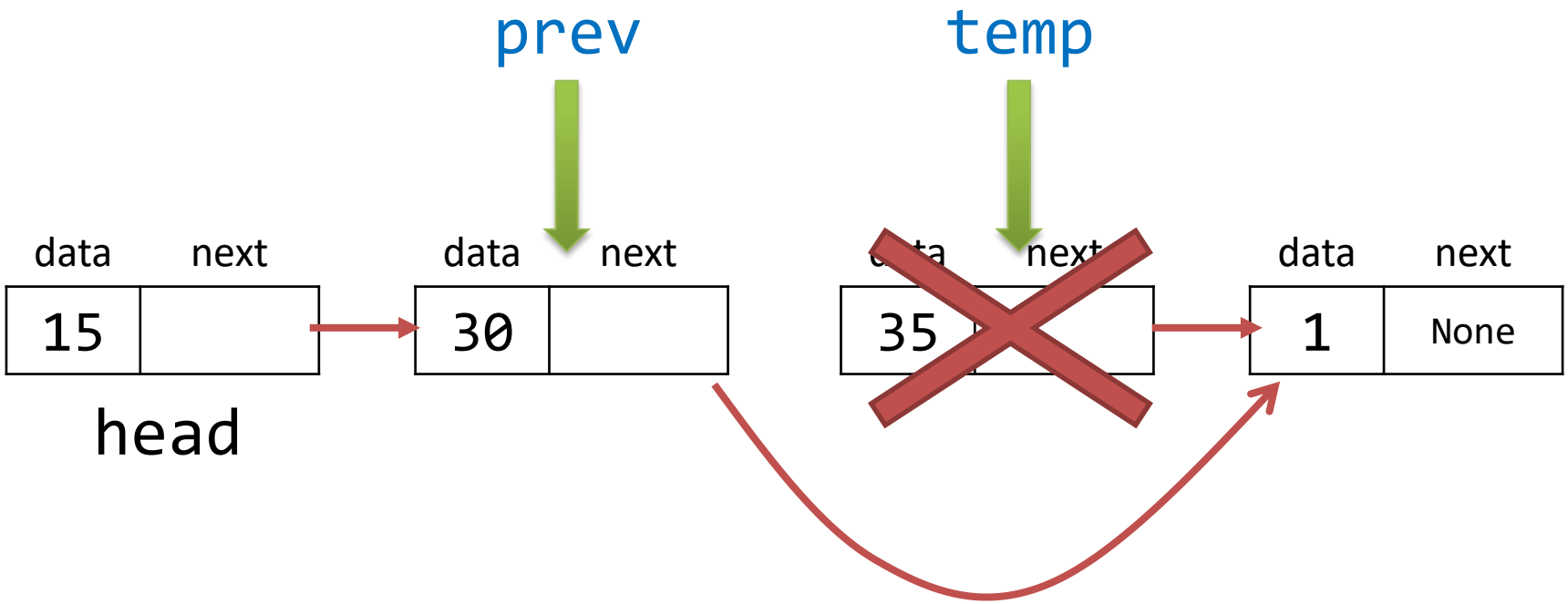
Delete algorithm




```
delete(value)
```

1. Start at the head node
2. Search for the node whose data = `value` and keep track of the `previous` node of temp
3. `prev.next = temp.next`
4. `Delete temp`

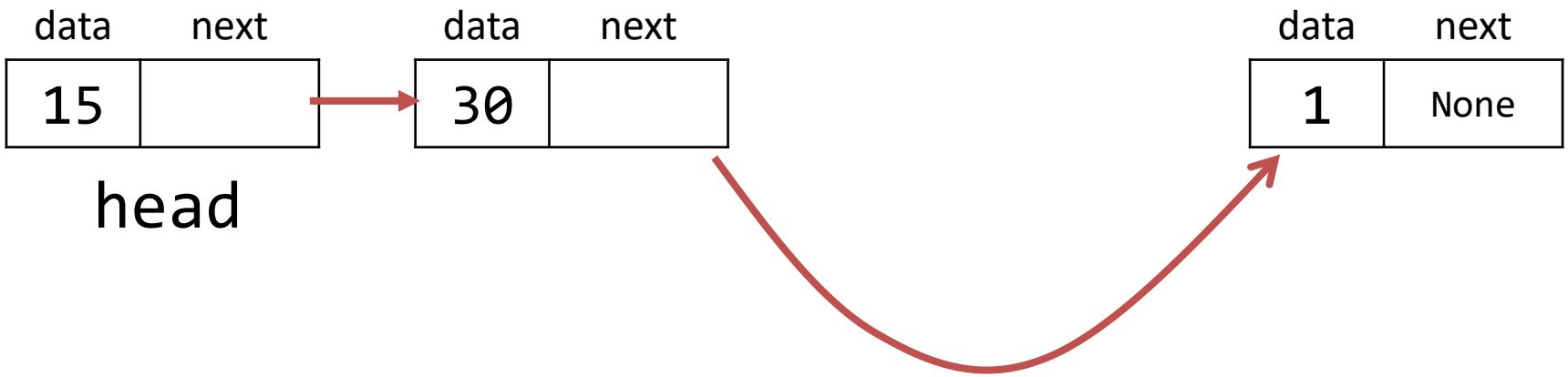
Delete algorithm




```
delete(value)
```

1. Start at the head node
2. Search for the node whose data = `value` and keep track of the `previous` node of temp
3. `prev.next = temp.next`
4. `Delete temp`

Delete algorithm-revised



Delete algorithm in Python

```
def delete(self, value):
    prev = None
    temp = self.head
    while temp is not None:
        #Value not found yet
        if temp.data is not value:
            prev = temp
            temp = temp.next
        #Value found
        else:
            #node to be deleted is head
            if temp == self.head:
                self.deleteAtHead()
            else:
                prev.next = temp.next
                del temp
            return
    print'Value ', value, ' cannot be found'
```

Singly linked list operations

- Other operations
  - `printList()`
  - `insert(node, index)`
    - insert `node` before/after `index`

Print list in Python

```
def printList(self):
    print("Current list content:")
    temp = self.head
    while temp is not None:
        print '[',temp.data,']',
        temp = temp.next
    print
```

Singly linked list - ADT

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def search(self, value):...
    def insertAtHead(self, node):...
    def delete(self, value):...
    def deleteAtHead(self):...
    def printList(self):...
```

Using Singly linked list - ADT

```
aList = SinglyLinkedList()
aNode = SinglyListNode(15)
aList.insertAtHead(aNode)
aNode2 = SinglyListNode(30)
aList.insertAtHead(aNode2)
aNode3 = SinglyListNode(40)
aList.insertAtHead(aNode3)
aList.printList()

aList.delete(30)
aList.printList()
aList.delete(3)
```

```
Current list content:
[ 40 ][ 30 ][ 15 ]
Current list content:
[ 40 ][ 15 ]
Value 3 cannot be found
```

Doubly Linked-list

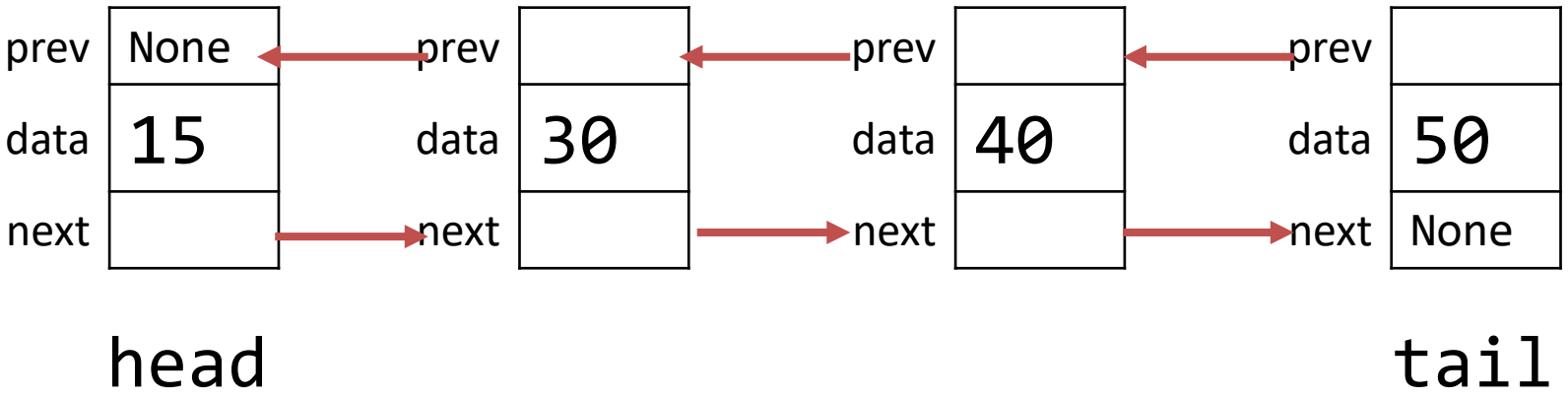
A doubly linked list is a `linear` data structure in which each element (node) consists of three items:

1. Data.
2. A reference (`link`) to the `next` node in the list.
3. A `link` to the `previous` node in the list.

Doubly linked list - ADT

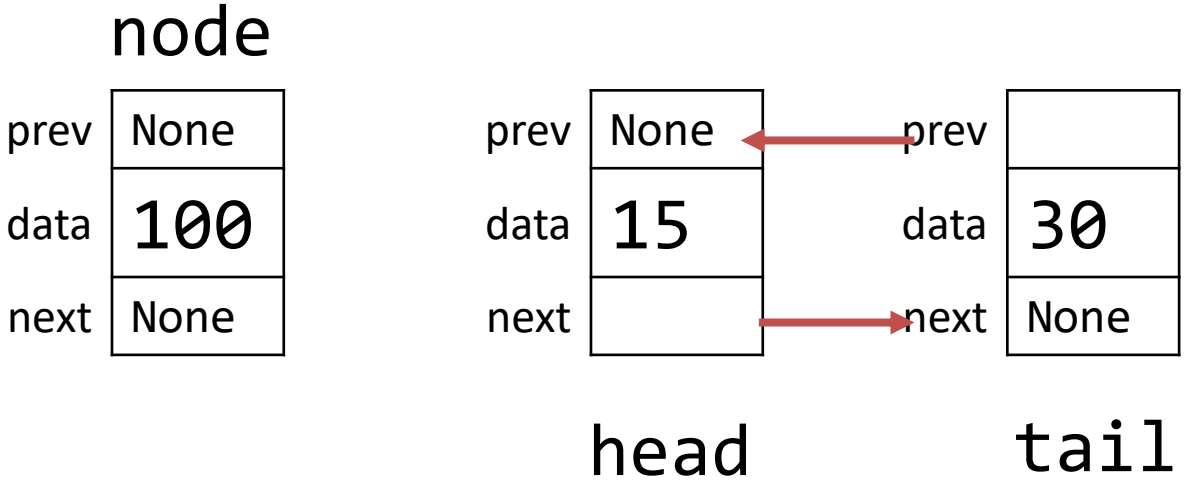
```
class DoublyListNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def search(self, value):...
    def insertAtHead(self, node):...
    def delete(self, value):...
    def deleteAtHead(self):...
    def printList(self):...
```

Doubly linked list visualization



DLL algorithms

```
insertAtHead(node)
```

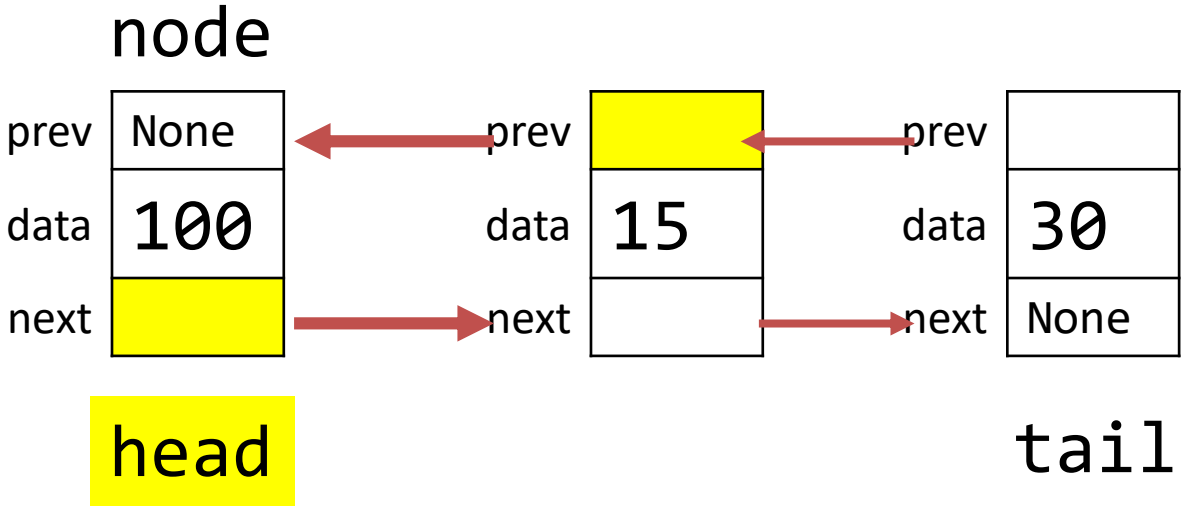



```
head.prev = node
node.next = head
head = node
```

What if the link list is empty?

DLL algorithms

```
insertAtHead(node)
```

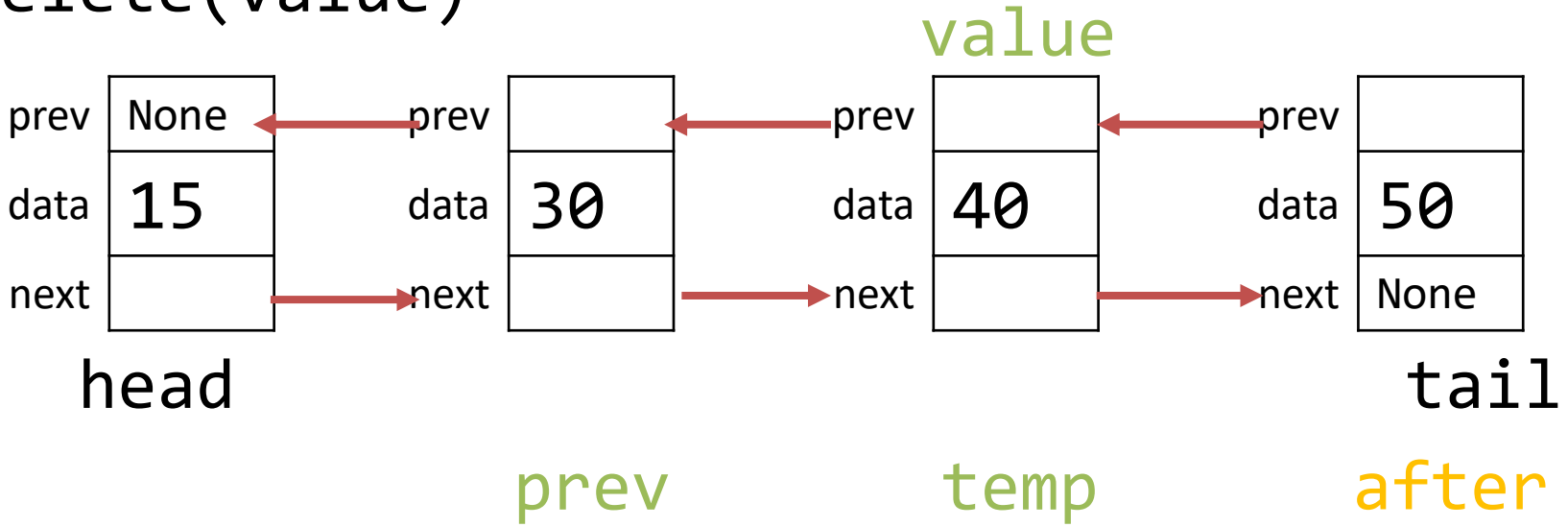


DLL algorithms

```
def insertAtHead(self,node):
    if self.head is None:
        self.head = self.tail = node
    else:
        self.head.prev = node
        node.next = self.head
        self.head = node
```

DLL algorithms

```
delete(value)
```

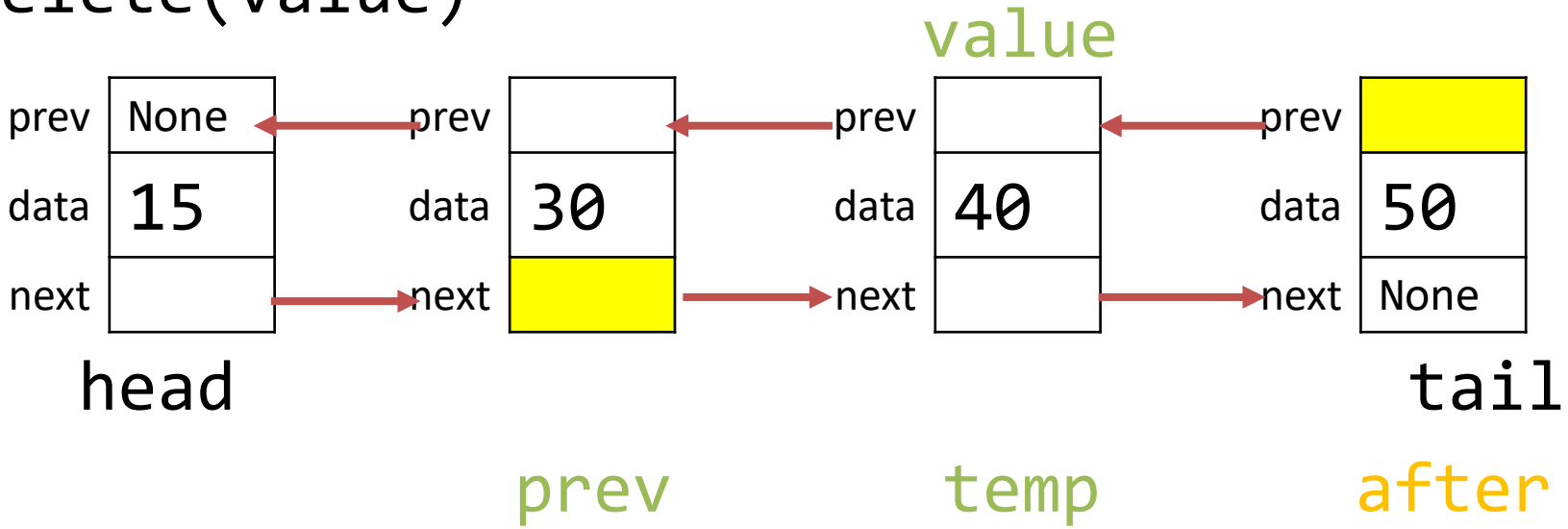


We do not need to keep track of `prev & after`.
They can be obtained easily by:

```
prev = temp.prev
after = temp.next
```

DLL algorithms

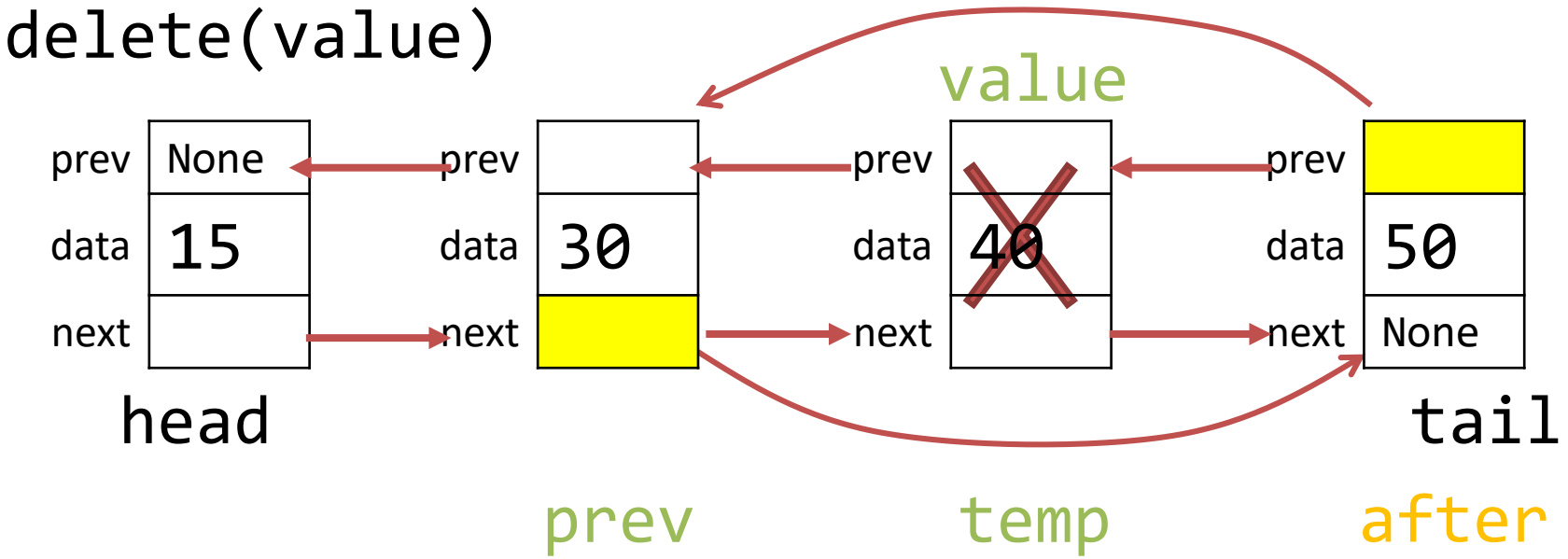
delete(value)



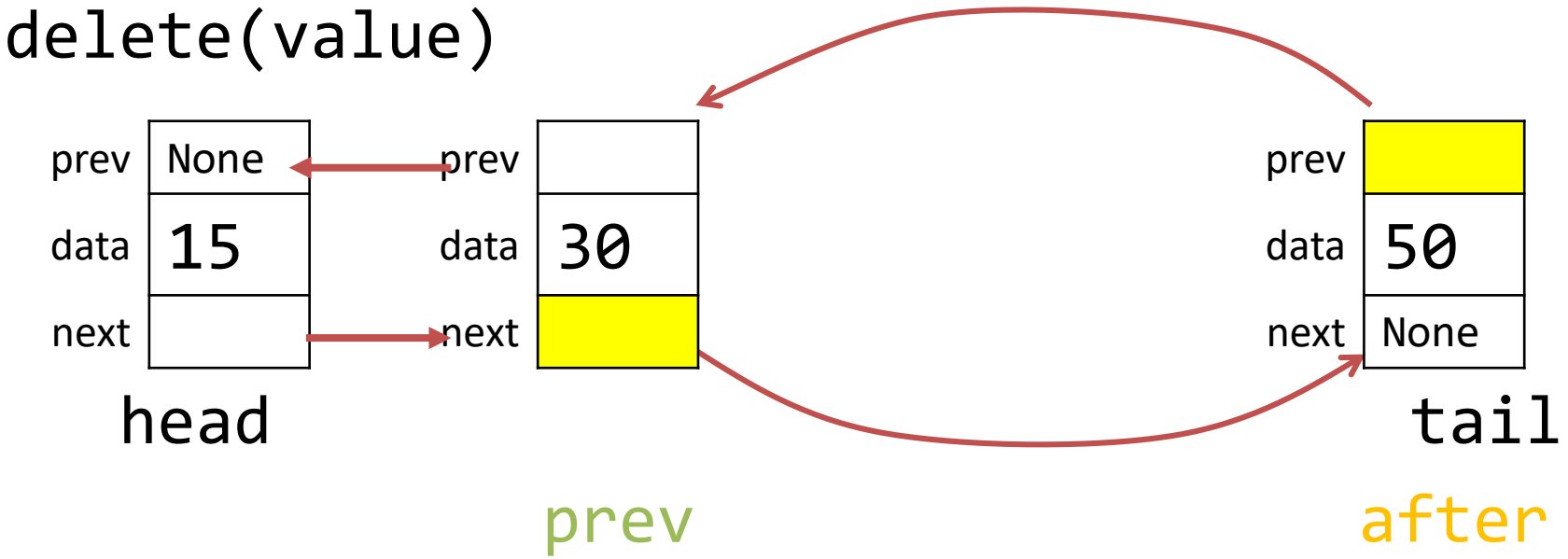

```
prev.next = after
after.prev = prev
del temp
```

How about `tail` & `head`?
Do we need to update `tail` & `head`?

DLL algorithms



DLL algorithms



DLL algorithms

```
def delete(self,value):
    temp = self.head
    while temp is not None:
        if temp.data is not value:
            temp = temp.next
        else:#Value found
            if temp is self.head: #delete at head
                self.head = self.head.next
            elif temp is self.tail: #delete at tail
                self.tail = self.tail.prev
            else: #normal delete case
                prev = temp.prev
                succ = temp.next
                prev.next = succ
                succ.prev = prev
            del temp
            return
    print 'Delete: Value not found'
```