CMPUT 175
Introduction to Foundations of Computing

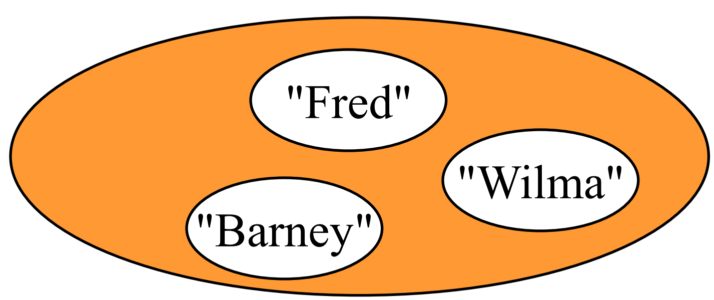
Singly-Linked List and
Doubly-Linked List

Objectives

- In this lecture we will learn about an implementation of Lists called Singly-Linked List
- We will first draw lists and discuss them.
- We will create a class for elements of a list then a class for the list itself.
- We will repeat the process for a Doubly-Linked List

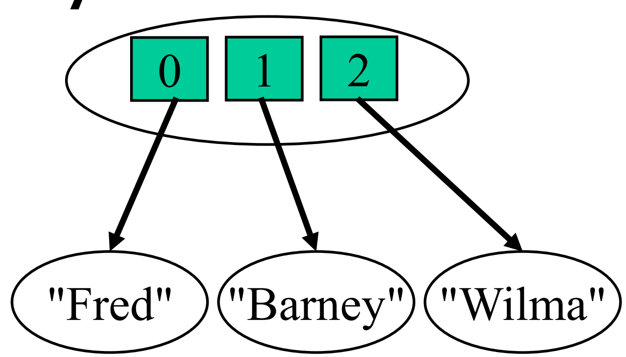
External Container Diagrams

- We can draw an external or implementation-independent diagram of a container by just showing its elements.
- For example, here is the diagram for a general container where the elements are not even ordered:



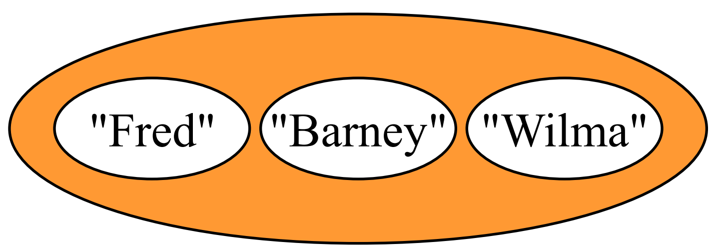
Indexed Container Diagrams

- We can modify the diagram when the interface is more specific.
- For example, here is a diagram for an indexed container.
- Note that it still might be implemented in different ways



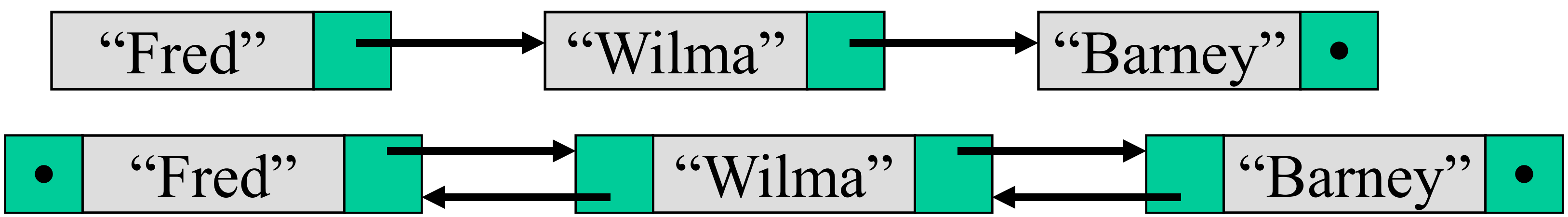
External List Diagrams

- Since the elements in a list are ordered, they must be “connected” somehow to maintain this order.
- Since different implementation classes “connect” the elements differently, we do not show the connections in an external diagram.

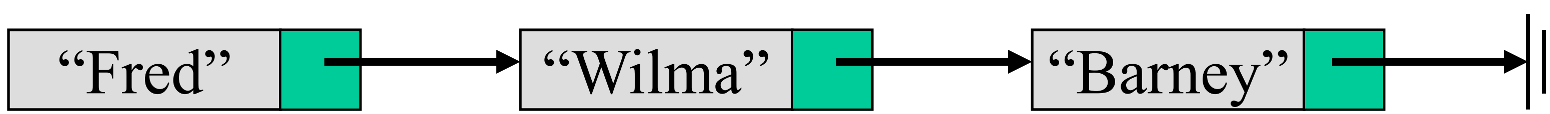


Internal List Diagrams - Nodes

- However, when we want to highlight a particular class implementation of a List, we add internal structure.
- Each element is put in a list **node** and the nodes connect to each other with **links**.
- The end of the list is denoted by a dot instead of a link to another node.

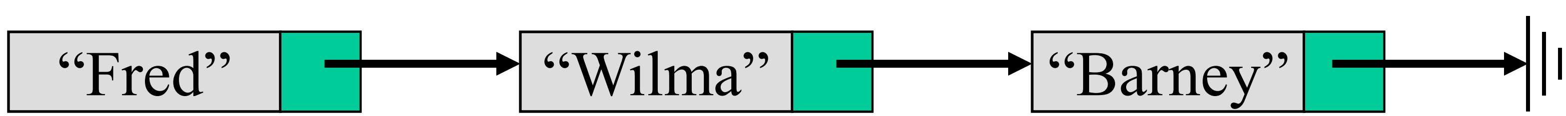


- Instead of a dot, it can be represented as a link going nowhere, sometimes called Nil, Null or None.



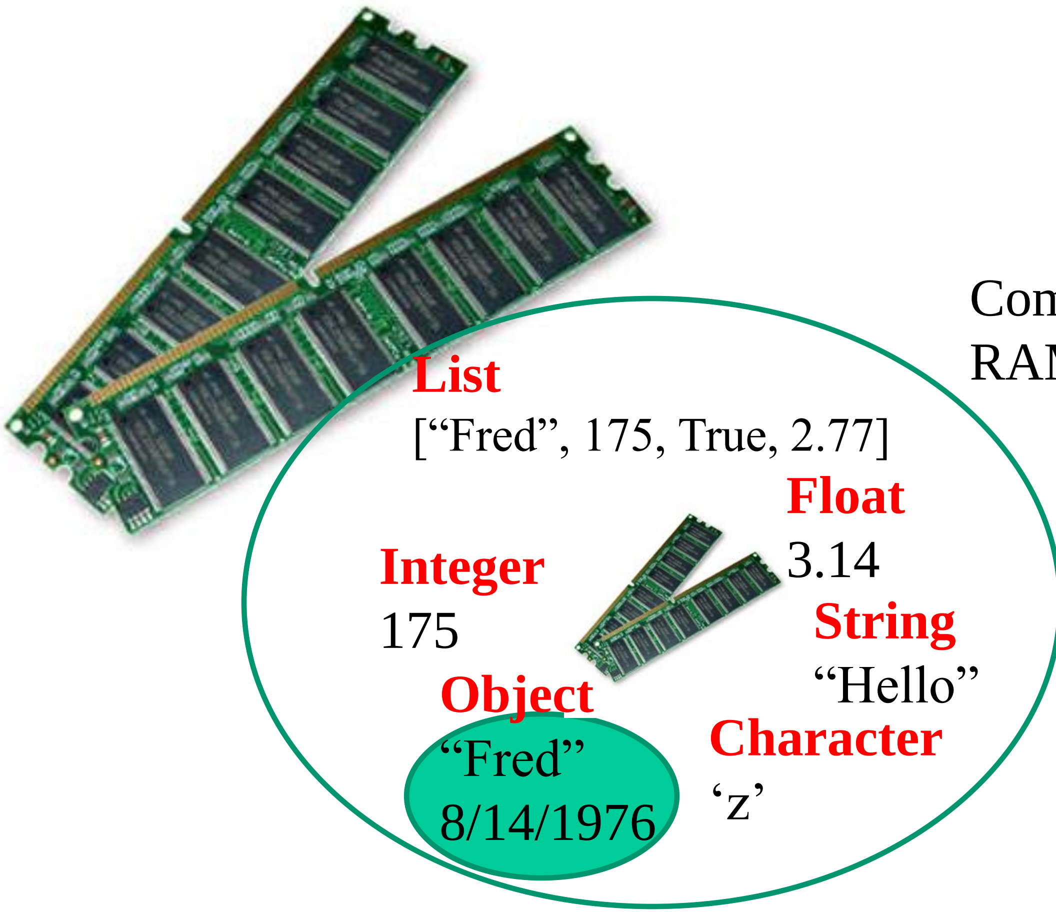
Links and Nodes

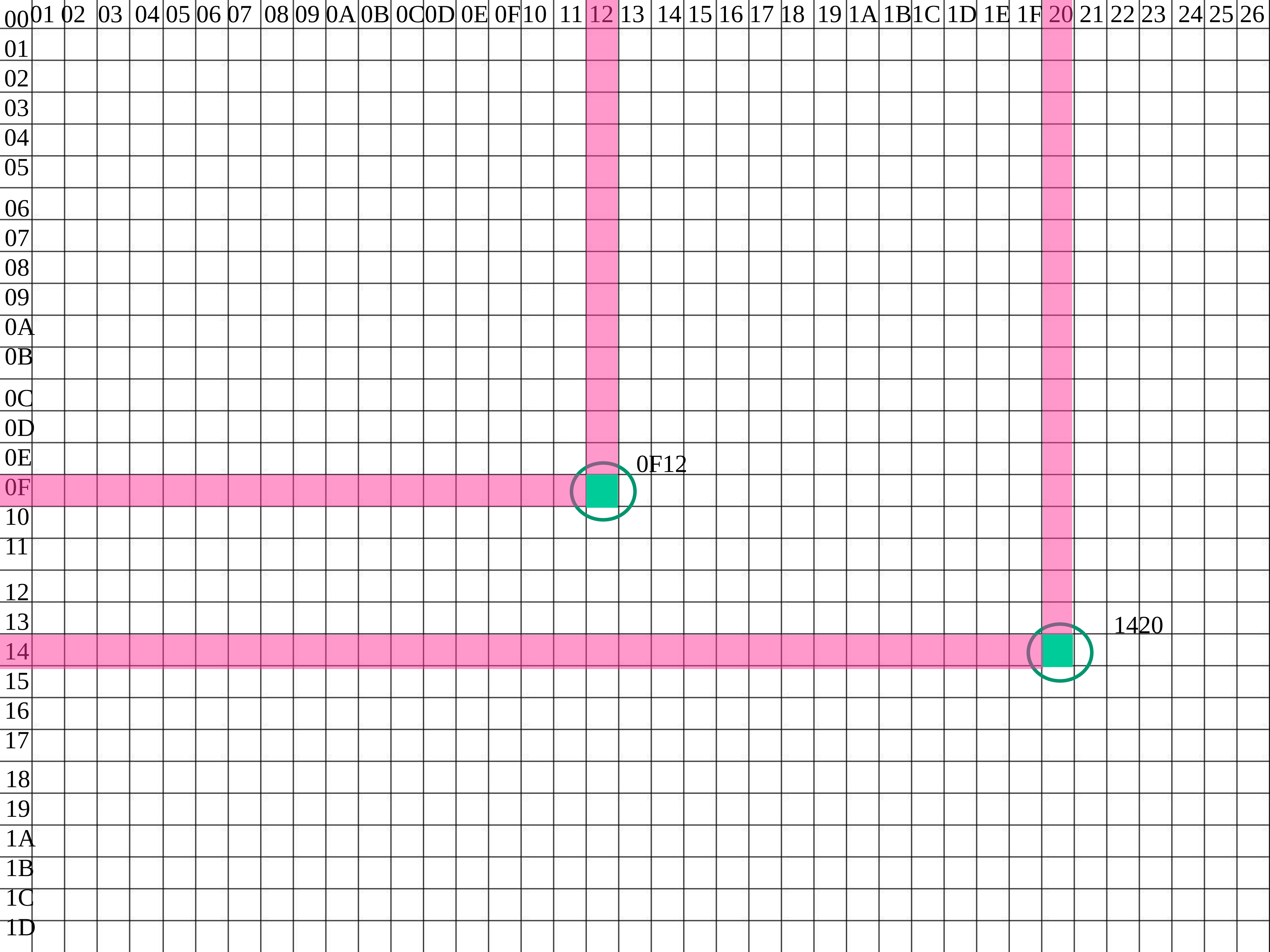
- Where are these nodes placed?
- What are these links from one node to the other?
- What can the content of these nodes be?



1 Byte = 8 bits = one characters
1 Kilobyte KB = 1024 bytes about 1000 bytes
1 Megabyte MB = about 1000 Kilobyte
1 Gigabyte GB = 1000 MB = about a billion bytes
1 Terabyte TB = 1000 GB

Computer Memory
RAM: Random Access Memory




00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26
01
02
03
04
05
06
07
08
09
0A
0B
0C
0D
0E
0F
10
11
12
13
14
15
16
17
18
19
1A
1B
1C
1D
0F12
1420

Character

c

Integer

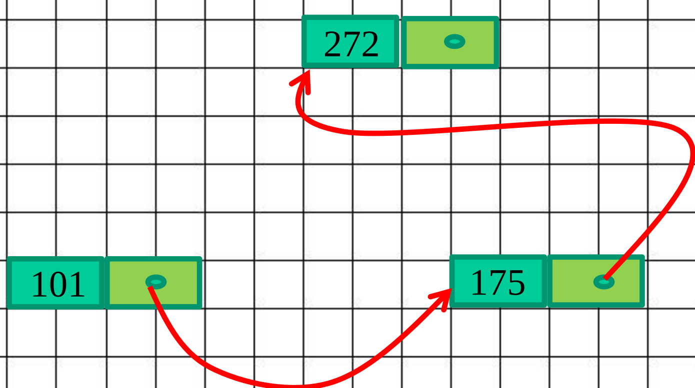
175

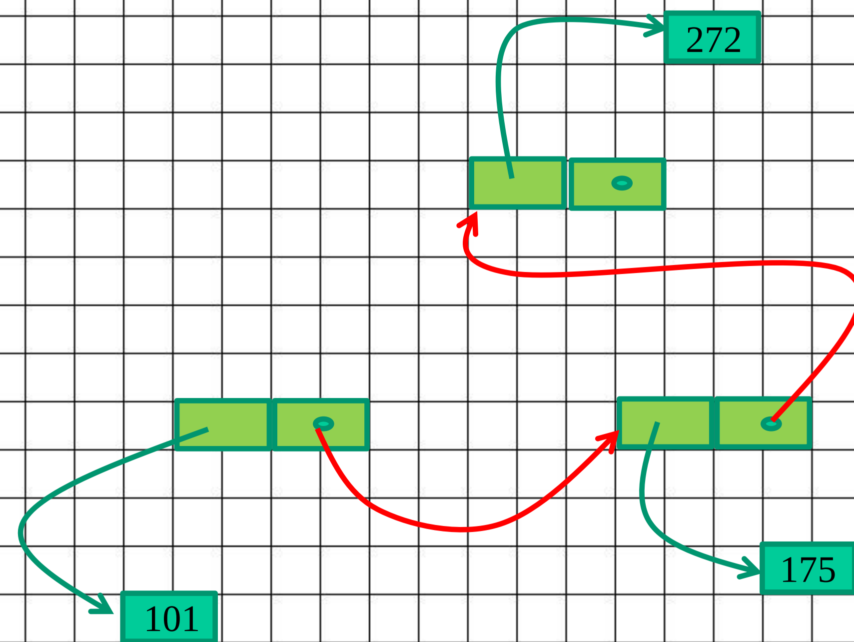
Long Integer

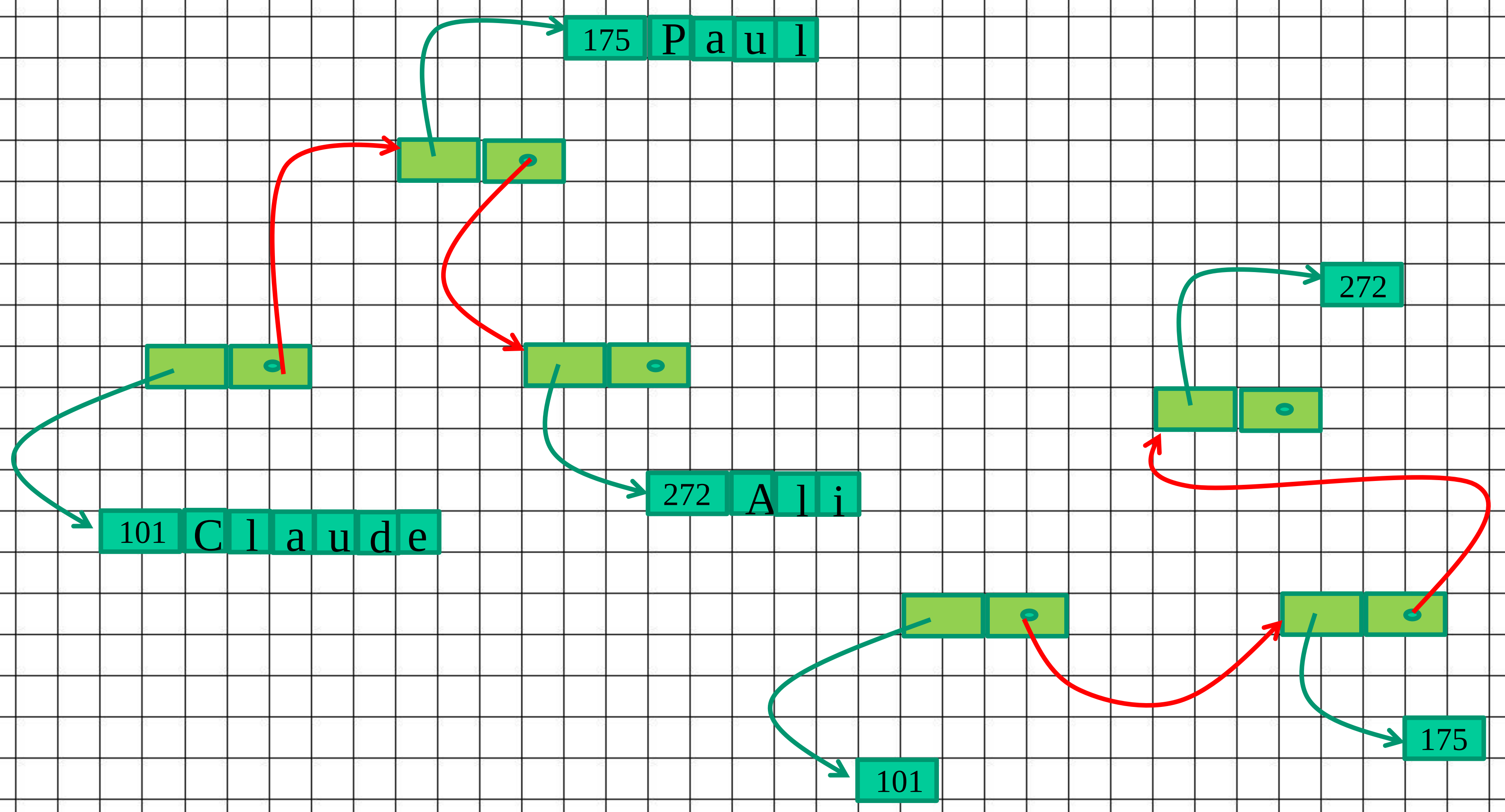
20170301

Array of 10 Integers

101	175	272	?	?	?	?	?	?	?



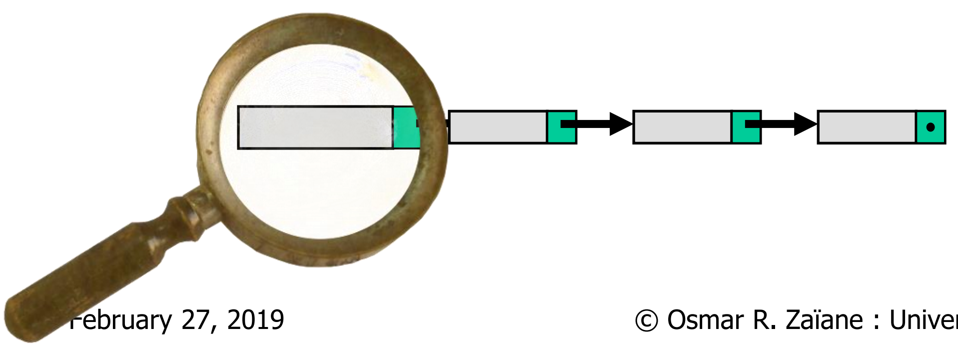


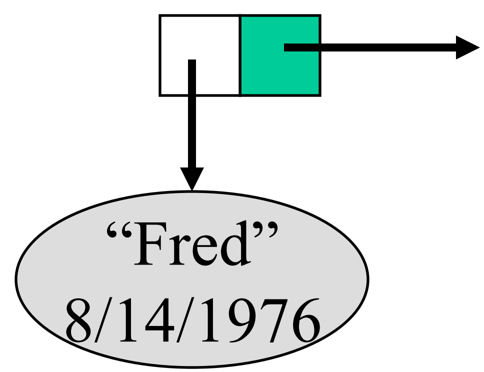


175 Paul
101 Claude
272 Ali
272
101
175

Complex List Node Diagrams

- If the elements of a list are complex objects, it is not always possible to draw the elements inside the node.
- In this case, an arrow is used in the node to represent a reference to the element.
- This diagram is actually more accurate since the node always contains a reference to an object instead of an object.

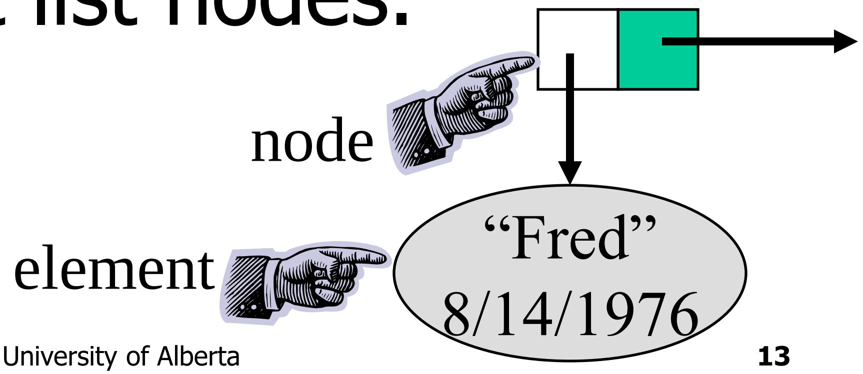


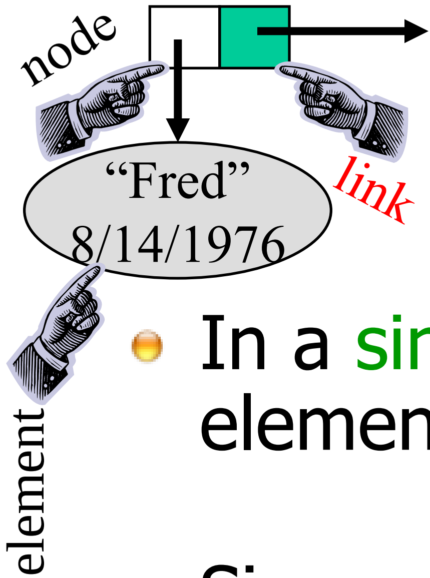


Terminology: Elements & Nodes

- In these notes we use the term **element** to refer to the individual values or objects that collectively make up the list.

- We use the term **node** to refer to an object that contains an element object and links to adjacent list nodes.



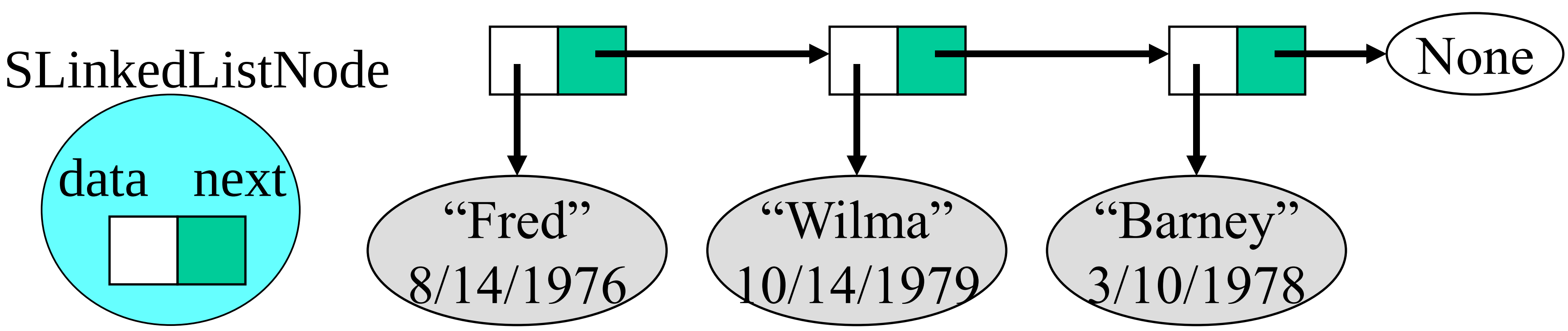


Singly-Linked Lists

- In a singly-linked list, each list node contains an element and a link to the "next" node in the list.
- Since a node contains a link to the next node, this is an example of self-referencing of objects. It points to another instance of the same class.
- We need to define two classes to implement a singly-linked list: one for the nodes and one for the list itself.
- We will call the first one **SLinkedListNode** and the second **SLinkedList**

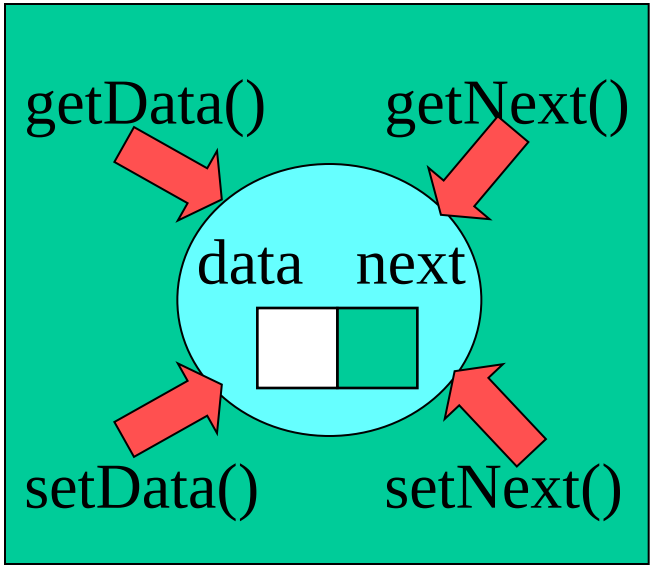
SLinkedListNode Class

- Each instance of SLinkedListNode represents a single list node with two instance variables.
- The instance variable, data, is a reference to an element object.
- The instance variable, next, is a reference to the next node (another instance of SLinkedListNode) or None if it is the last node (tail node).



Interface for ADT Node

- setData(element) set element as the new data
- setNext(reference) set reference as the new next
- getData() returns the data element
- getNext() returns the reference to the next node



SLinkedListNode in Python

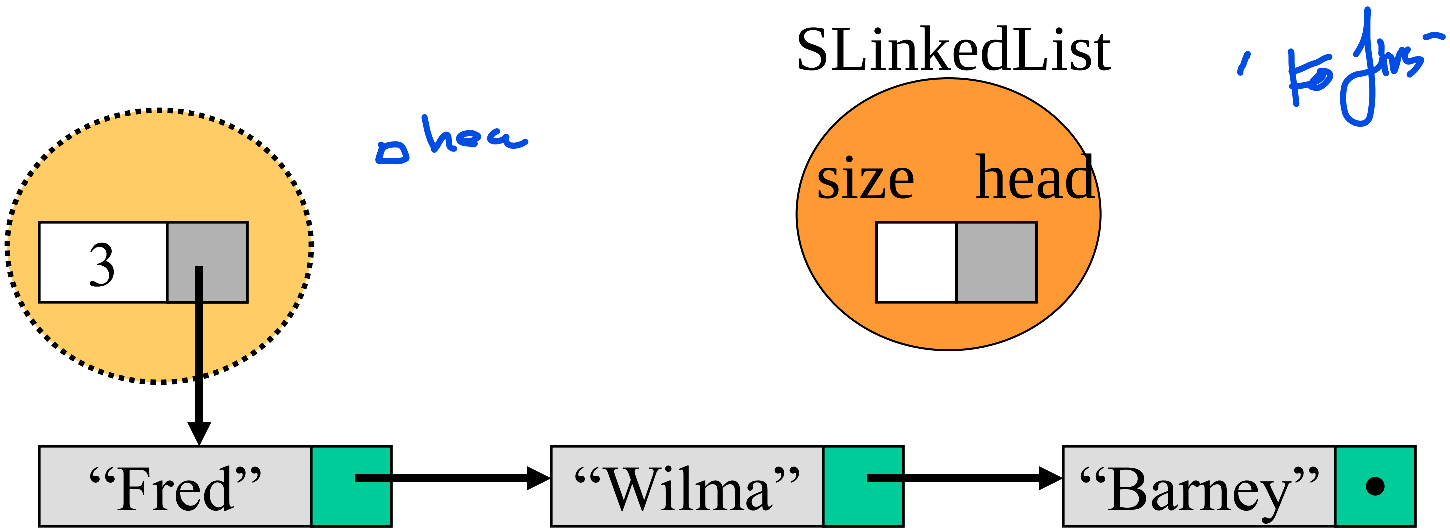
```
class SLinkedListNode:
    def __init__(self, initData, initNext):
    # constructs a new node and initializes it to contain
    # the given object (initData) and link to the given next node.

        self.data = initData
        self.next = initNext


    def getData(self): # returns the element
        return self.data

    def getNext(self): # returns the next node
        return self.next

    def setData(self, newData): # sets the newData as the element
        self.data = newData

    def setNext(self, newNext): # sets the newNext as the next node
        self.next= newNext
```

Interface for ADT List

- add(item) adds a new item to the list
- remove(item) removes the item from the list
- search(item) returns a boolean value if item in list
- isEmpty() tests to see whether the list is empty
- length() returns the number of elements in the list
- append(item) adds item to the end of list
- index(item) returns the position of item in the list
- insert(pos,item) adds an item at a given position in list
- pop() removes and returns the last item
- pop(pos) removes and returns the item at a position

SLinkedList Class

- Each SLinkedList object has an instance variable, head, that is a reference to the first SLinkedListNode object of the list.
- It also maintains an integer valued instance variable, size, that is the size of the list.

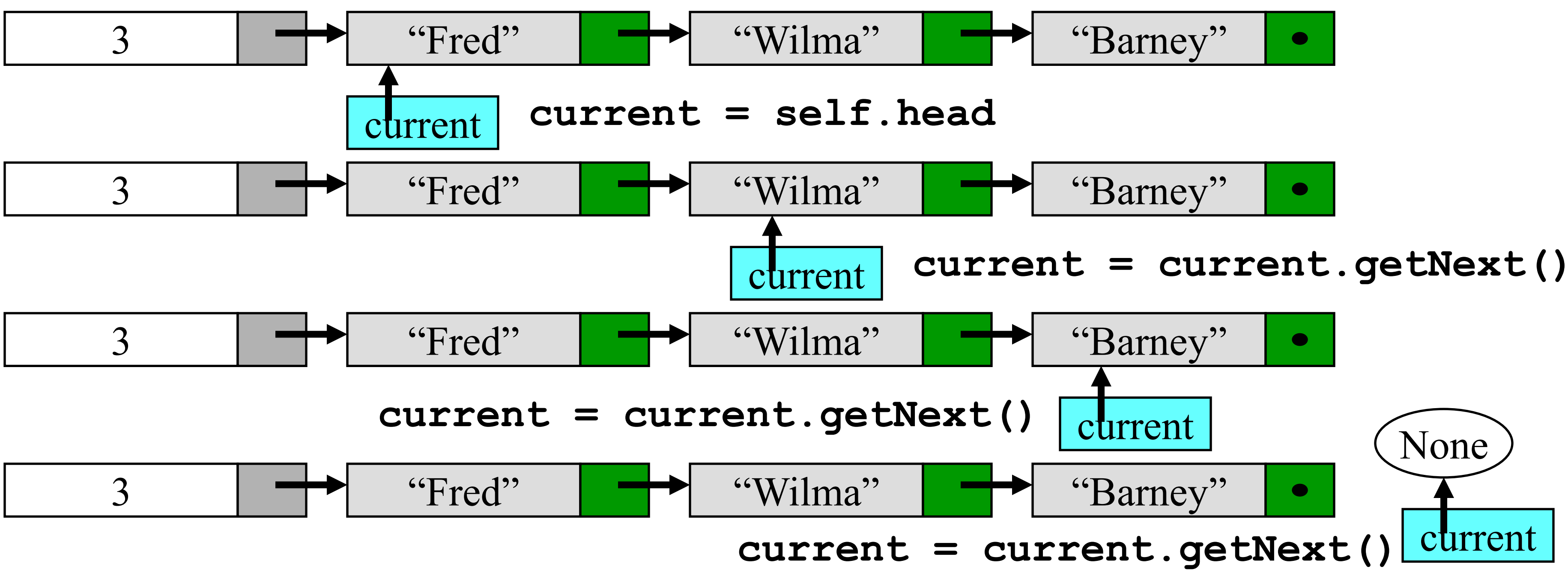


Why Caching the List Size

- The list size could be computed by traversing the list and counting nodes.
- However, the size is cached as an instance variable so that the size() method can be computed faster.
- The disadvantage of caching the size as an instance variable is that the instance variable must be updated each time an element is added or removed from the list.

List Traversal

- Many list methods will involve traversal of the list elements from the head to the tail or from the head to a particular element or location.
- We use a cursor (called current) for traversal.



SLinkedList in Python

```
class SLinkedList:
    def __init__(self):
        self.head=None
        self.size=0

    def isEmpty(self):
        return self.head == None

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

```
def length(self):
    current = self.head
    count = 0
    while current != None:
        count = count + 1
        current = current.getNext()

    return count
```

If we do not cache the size we could traverse the list and count the elements

Converting to a string

- Before we see the implementations of the other methods in the public interface, we can see who to convert the list to allow it to be printed.
- We simply traverse the list and build a string that contains the elements and put them in between “[“ and “]”

```
def __str__(self):
    s= '['
    i=0
    current=self.head
    while current != None:
        if i>0:
            s = s + ','
        dataObject = current.getData()
        if dataObject != None:
            s = s + "%s" % dataObject
            i = i + 1
        current = current.getNext()
    s = s + ']'
    return s
```

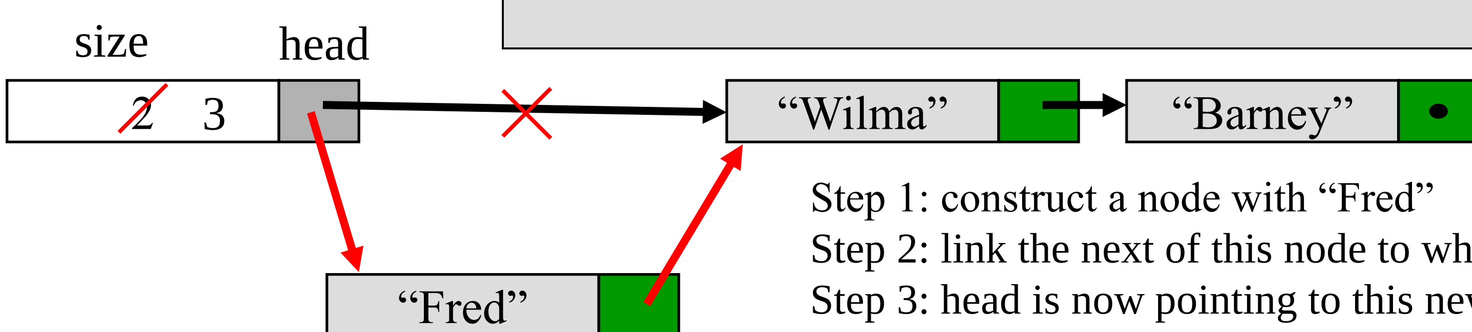
SLinkedList – add(item)

```
def add(self, item):
    # adds an item to list at the beginning

    temp = SLinkedListNode(item,None)
    temp.setNext(self.head)
    self.head = temp
    self.size += 1
```

We could also have done the following:

```
def add(self, item):

    temp = SLinkedListNode(item,self.head)
    self.head = temp
    self.size += 1
```

add(“Fred”)

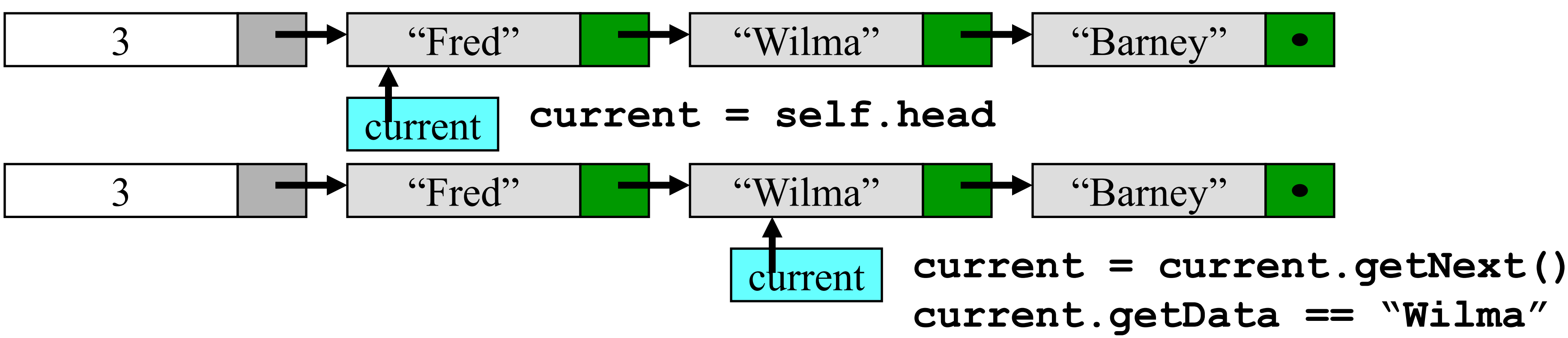


Step 1: construct a node with “Fred”
Step 2: link the next of this node to what head points to
Step 3: head is now pointing to this new node
Step 4: update size

SLinkedList – search(item)

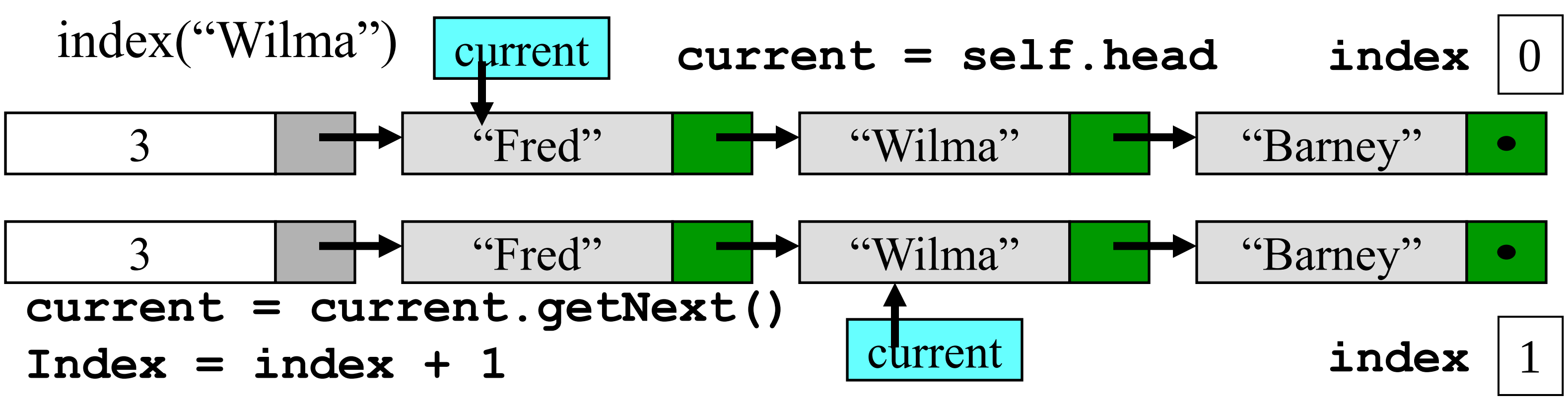
```
def search(self, item):
    current = self.head
    found = False
    while current != None and not found:
        if current.getData() == item:
            found= True
        else:
            current = current.getNext()

    return found
```

search("Wilma")



SLinkedList – index(item)

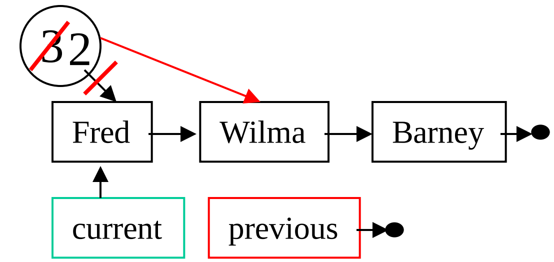
```
def index(self, item):
    # searches for the item and returns its order
    # number(index). Returns -1 if item doesn't exist
    current = self.head
    found = False
    index = 0
    while current != None and not found:
        if current.getData() == item:
            found= True
        else:
            current = current.getNext()
            index = index + 1
    if not found:
        index = -1
    return index
```



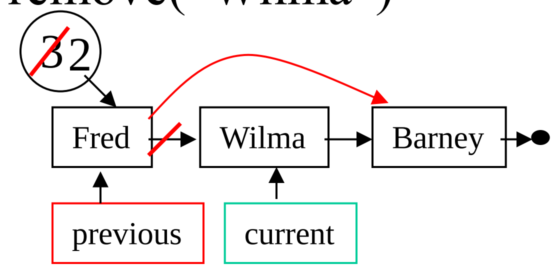
SLinkedList – remove(item)

```
def remove(self, item):
    # searches for the item and removes it
    # the method assumes the item exists
    current = self.head
    previous=None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()
    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
    self.size -= 1
```

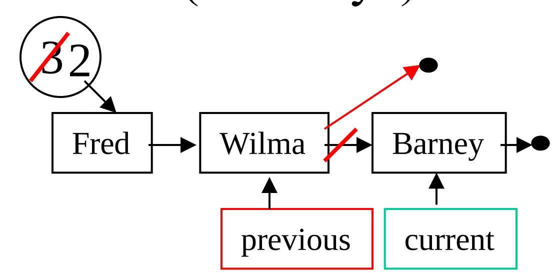
remove("Fred")



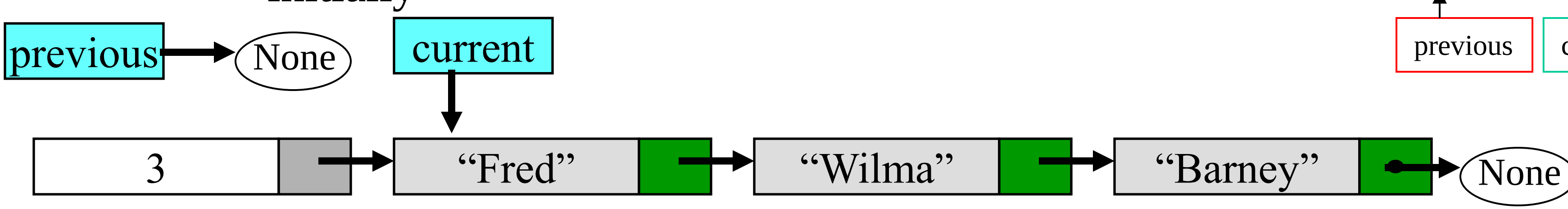
remove("Wilma")



remove("Barney")



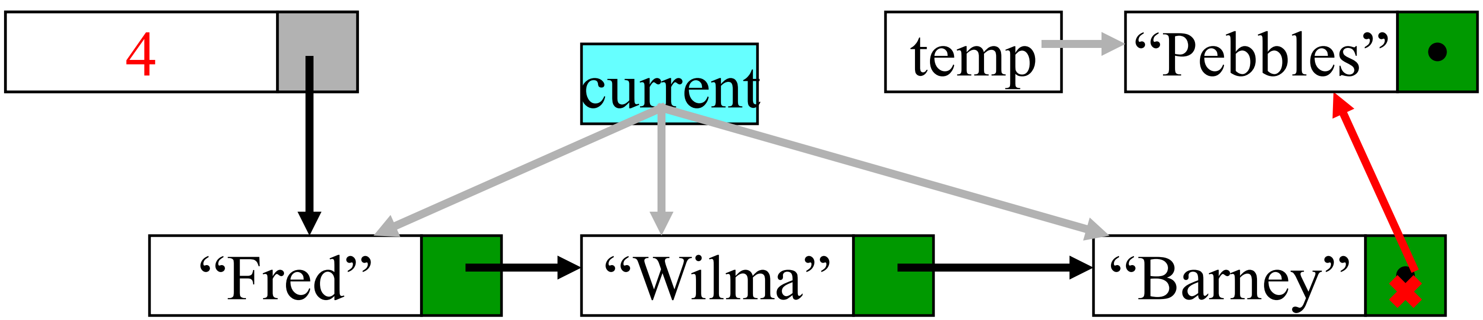
initially



SLinkedList - append(item)

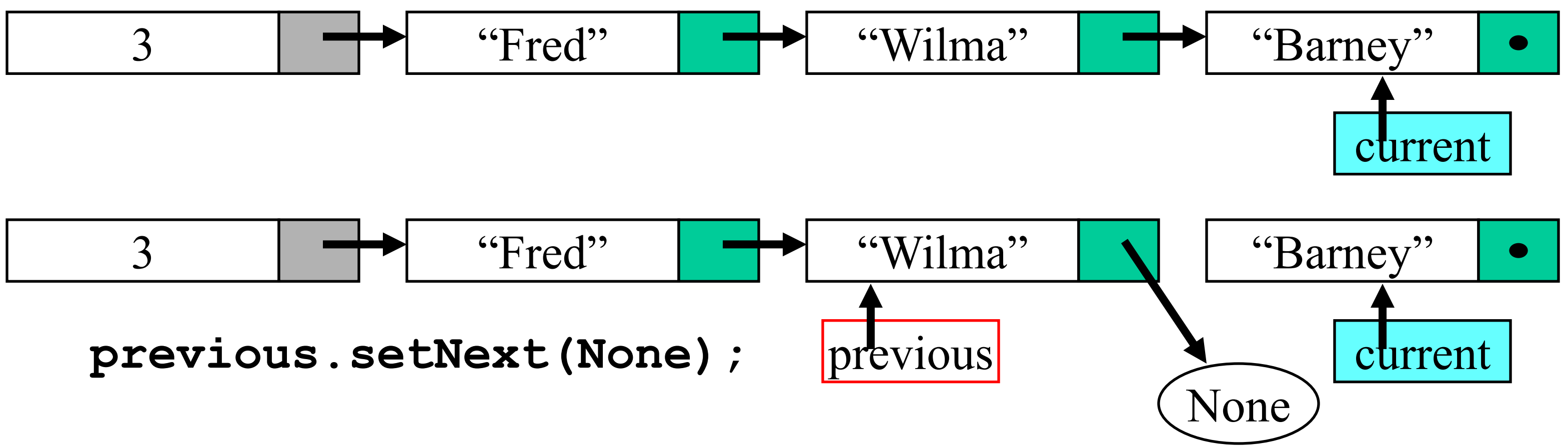
```
def append(self, item):
    # adds the item to the end of the list
    # must traverse the list to the end and add item
    temp = SLinkedListNode(item, None)
    if (self.head == None):
        self.head=temp
    else:
        current = self.head;
        while (current.getNext() != None):
          current = current.getNext()
        current.setNext(temp)
    self.size +=1
```

append("Pebbles")



Pop()=Removal From Tail

- We cannot remove from the tail by just traversing to the last node and removing it.
- We need a reference to the second last node so we can set its “next” reference to None.

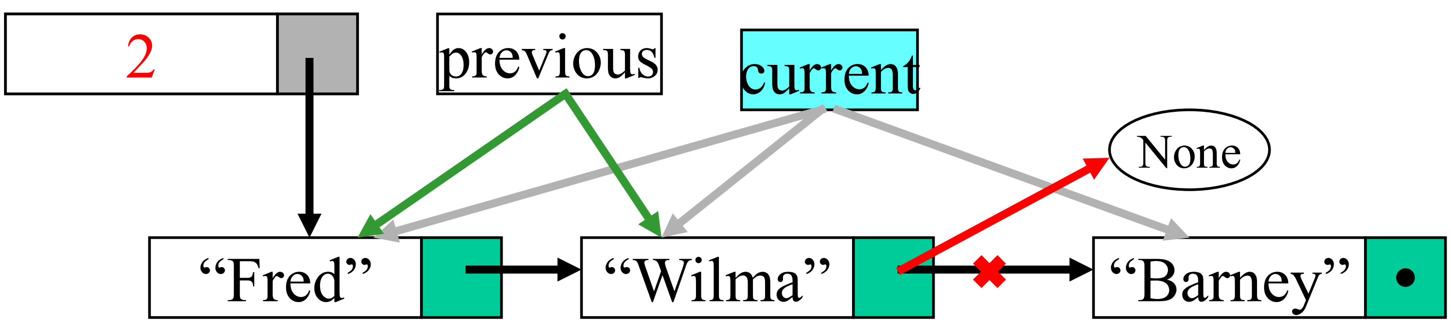



```
previous.setNext(None);
```

- To find the second last node, we need to traverse the list with a second cursor following the first.

SLinkedList - pop()

```
def pop(self):

    current = self.head
    previous = None
    while (current.getNext() != None):
        previous = current
        current = current.getNext()

    if (previous == None):
        self.head = None
    else:
        previous.setNext(None)
    self.size -= 1
    return current.getData()
```



The rest of the methods

- pop(pos) removes and returns the item at a position

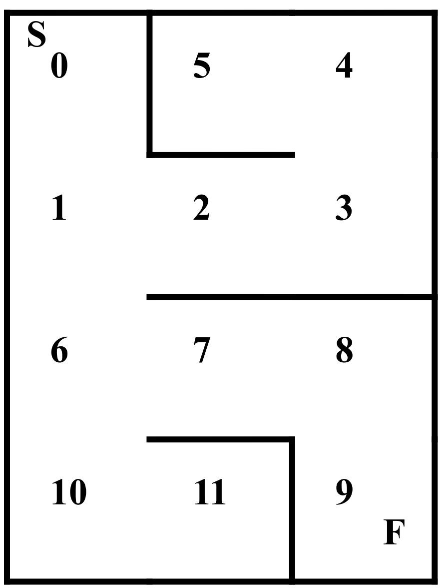
Left as an exercise
There could be other methods such as peek(), clear(), etc.

Linked-List Implementation Advice

- When manipulating references, draw pictures.
- Every public method of an object should leave the object in a consistent state.
- Test the boundaries of your structures and methods.

Linked List for MAZE

- Recall the problem of MAZE traversal from one of the earlier classes.
- We will show here how neighboring positions can be stored using a Linked List.



0 : {1}
1 : {0, 2, 6}
2 : {1, 3}
3 : {2, 4}
4 : {3, 5}
5 : {4}
6 : {1, 7, 10}
7 : {6, 8}
8 : {7, 9}
9 : {8}
10 : {6, 11}
11 : {10}

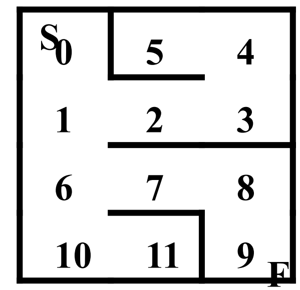
Creating an Adjacency List of Linked Lists

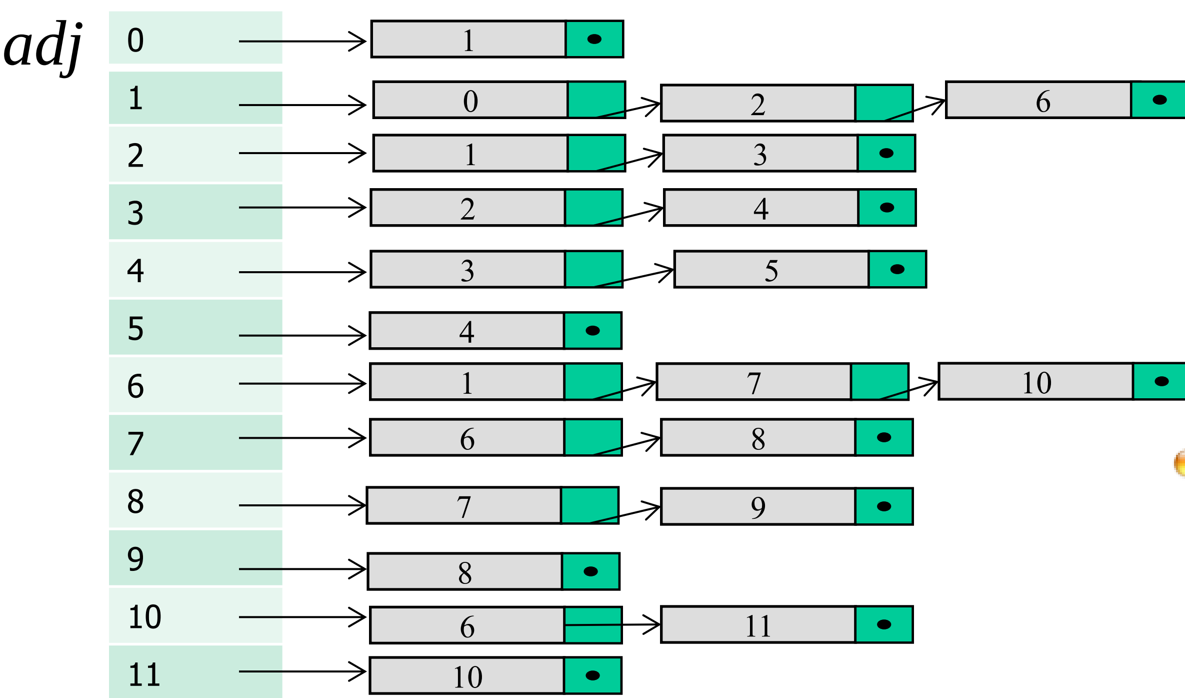
- First create an array of null pointers:

```
adj = []
for i in range(12): #create 0..11 empty lists
    adj.append(None)
```

adj

0	NONE
1	NONE
2	NONE
3	NONE
4	NONE
5	NONE
6	NONE
7	NONE
8	NONE
9	NONE
10	NONE
11	NONE





- Next create linked lists of legal positions that are adjacent to each position.

Creating an Adjacency List of Linked Lists

- We also need an array "visited" to store whether a state has been visited already

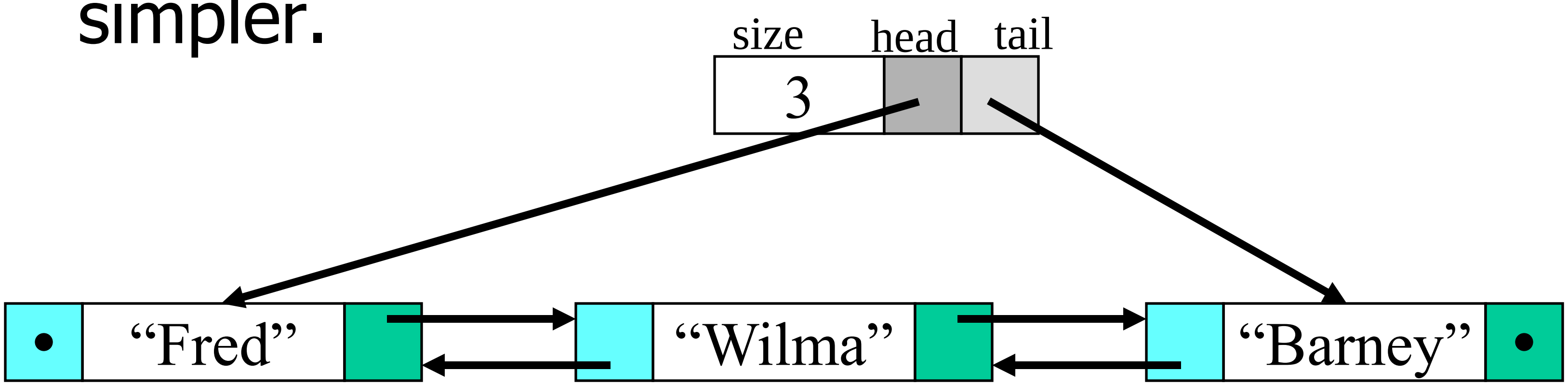
```
visited = []
for i in range(12): #initialize to False, True when visited
    visited.append(False)
```

Solving MAZE problem with Adjacency List & STACK

- With an Adjacency List, the order in which we select the Next Position to Visit depends on the order in which the Positions appear in an adjacency list.
- Recall that with a Stack we go as deep as possible (Depth First) before backing up. The sequence to Depth First Paths we take depends on the order of nodes in the adjacency lists.
- The advantage of the linked list is to insert elements in the beginning of the list in constant time.

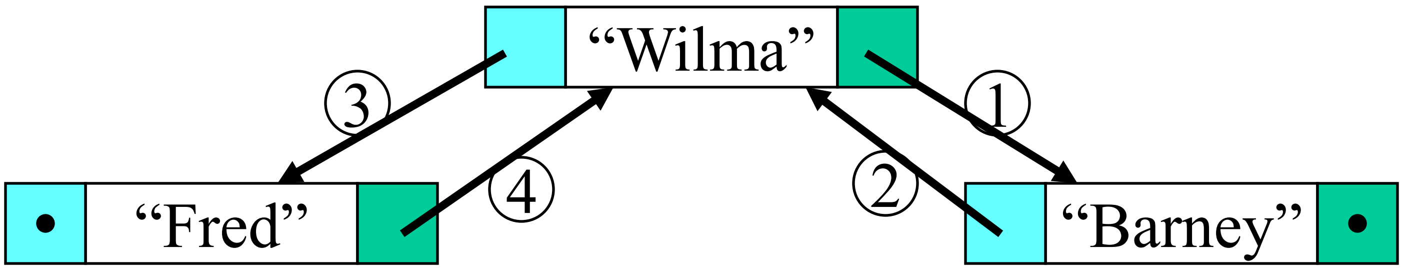
Doubly-Linked List Diagrams

- A doubly-linked list node has two links, one forward and one backward.
- The doubly-linked list has references to its head and tail nodes.
- This symmetry makes the implementation simpler.

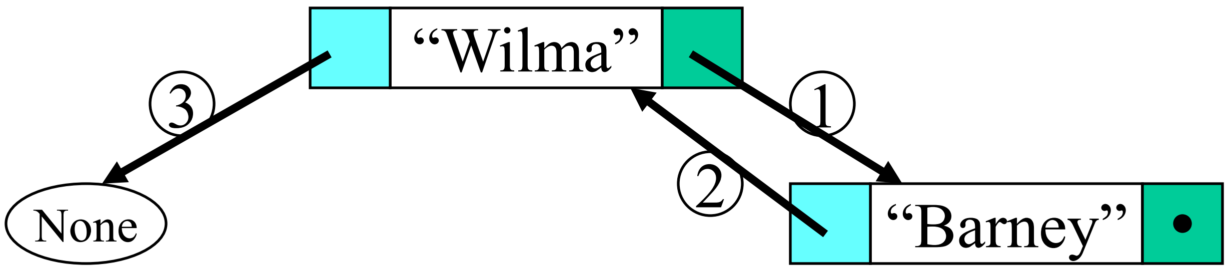


Constructing a Node

- When a DoublyLinkedListElement (node) is constructed, four links may need to be set.

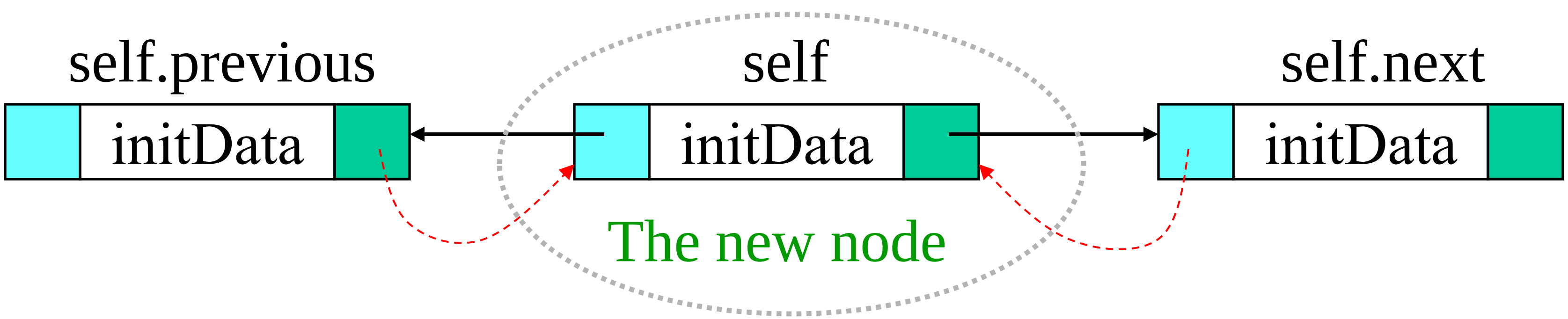


- If one or both of the “neighbouring” nodes is null then fewer links must be set.



DLinkedListNode in Python

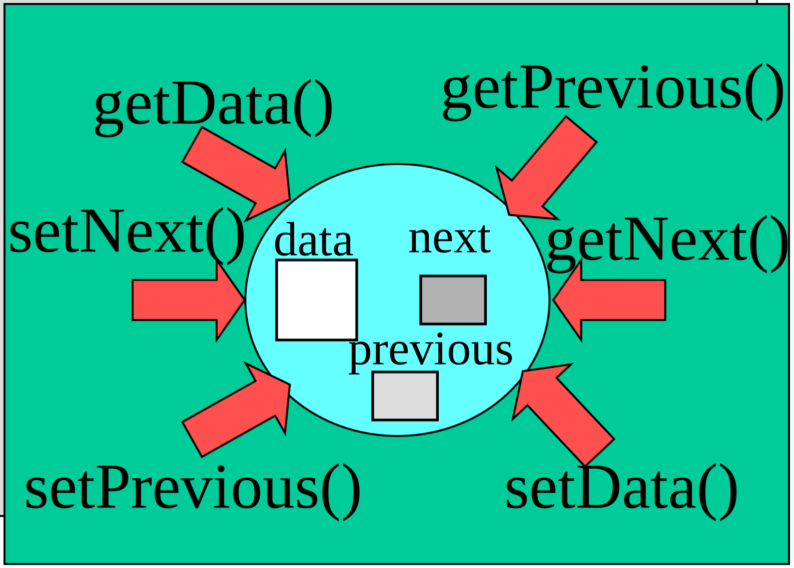
```
class DLinkedListNode:
    def __init__(self, initData, initNext, initPrevious):
    # constructs a new node and initializes it to contain
    # the given object (initData) and links to the given next
    # and previous nodes.

        self.data = initData
        self.next = initNext
        self.previous = initPrevious

        if (initPrevious != None):
            initPrevious.next = self

        if (initNext != None):
            initNext.previous = self
```



DLinkedListNode in Python

```
def getData(self):
    return self.data

def getNext(self):
    return self.next

def getPrevious(self):
    return self.previous


def setData(self, newData):
    self.data = newData

def setNext(self, newNext):
    self.next= newNext


def setPrevious(self, newPrevious):
    self.previous= newPrevious
```

Straightforward



DLinkedList in Python

```
class DLinkedList:

    def __init__(self):
        self.head=None
        self.tail=None
        self.size=0




    def isEmpty(self):
        return self.size == 0

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

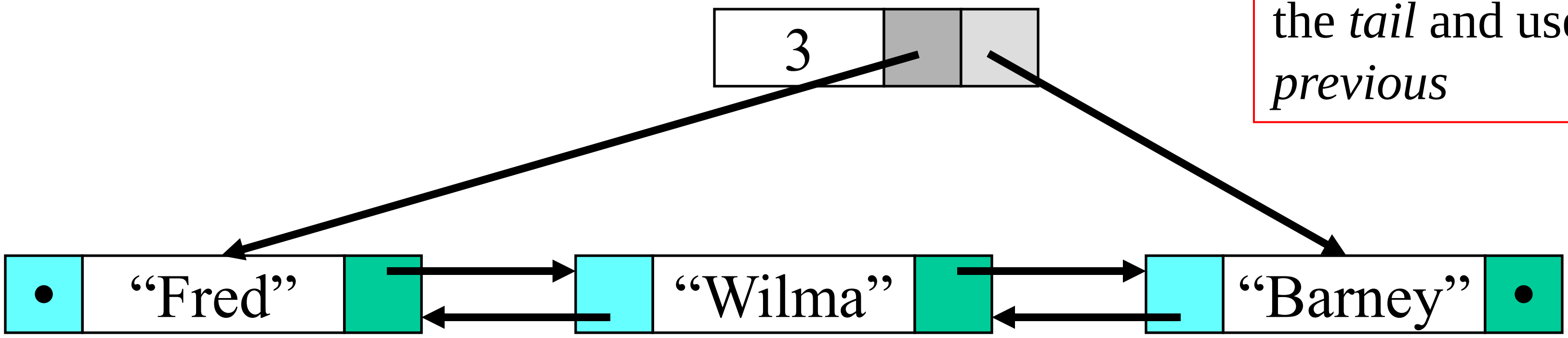
self.head==None

DLinkedList – search(item)

```
def search(self, item):
    current = self.head
    found = False
    while current != None and not found:
        if current.getData() == item:
            found= True
        else:
            current = current.getNext()

    return found
```

Identical to the search method for SinglyLinkedList

Note that we could also start the traversal from the *tail* and use *previous*

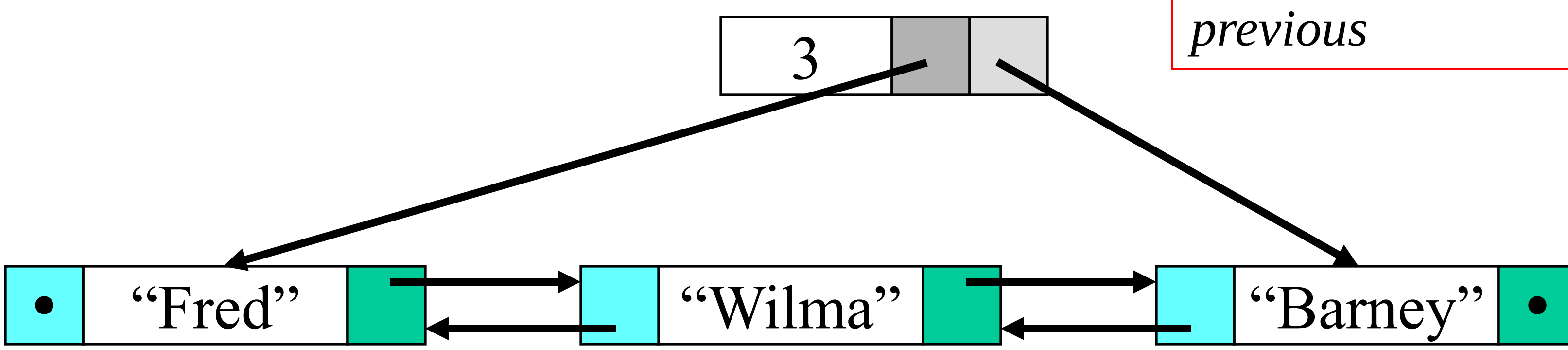


DLinkedList – index(item)

```
def index(self, item):
    current = self.head
    found = False
    index = 0
    while current != None and not found:
        if current.getData() == item:
            found= True
        else:
            current = current.getNext()
            index = index + 1
    if not found:
       index = -1
    return index
```

Identical to the index method for SinglyLinkedList

Note that we could also start the traversal from the *tail* and use *previous*

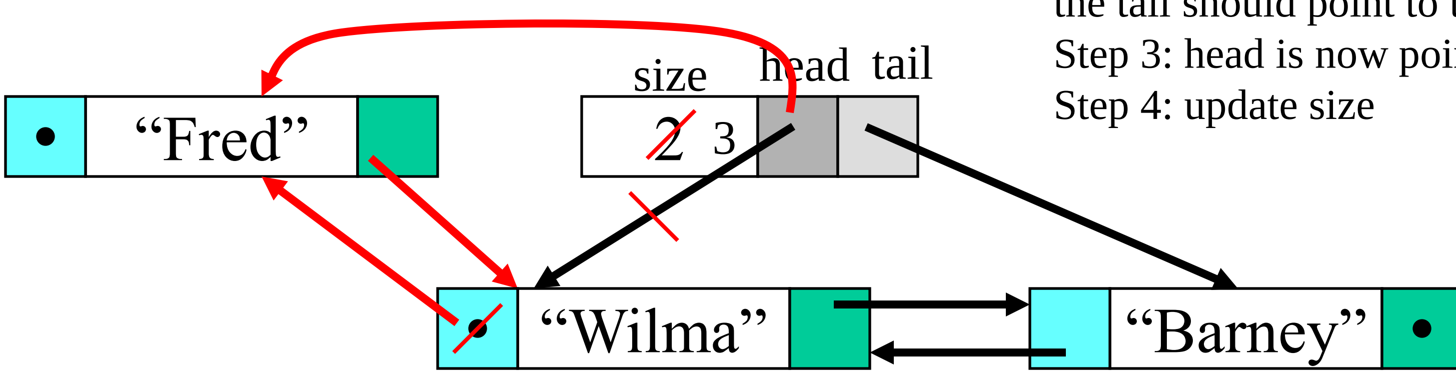


DLinkedList – add(item)

```
def add(self, item):
    # adds an item to list at the beginning

    temp = DLinkedListNode(item, self.head, None)
    if self.head != None:
       self.head.setPrevious(temp)
    else:
       self.tail=temp
    self.head = temp
    self.size += 1
```

add("Fred")

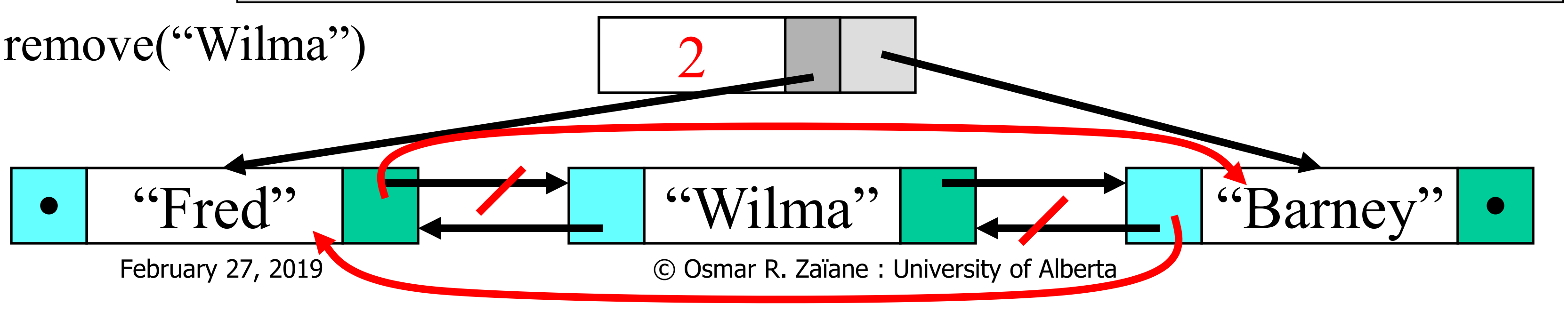
Step 1: construct a new node with "Fred"
Step 2: make the previous of old head to point to new node if the list isn't empty otherwise the tail should point to the new node
Step 3: head is now pointing to this new node
Step 4: update size



DLinkedList – remove(item)

```
def remove(self, item):
    # search for the item and remove it
    # the method assumes the item exists
    current = self.head
    previous=None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()
    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())

    if (current.getNext() != None):
        current.getNext().setPrevious(previous)
    else:
        self.tail=previous
    self.size -= 1
```

remove("Wilma")

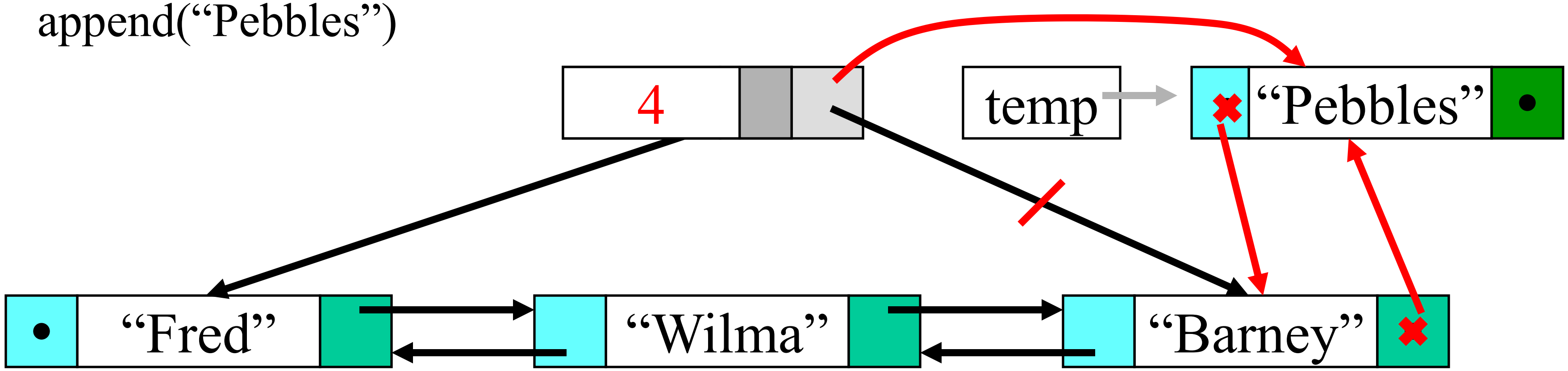


DLinkedList - append(item)

```
def append(self, item):
    # adds the item to the end of the list
    # must traverse the list to the end and add item
    temp = DLinkedListNode(item, None, None)
    if (self.head == None):
        self.head=temp
    else:
        self.tail.setNext(temp)
        temp.setPrevious(self.tail)

    self.tail=temp
    self.size +=1
```

There is no need for traversal

append("Pebbles")



The rest of the methods

- insert(pos,item) adds an item at a given position in list
- pop() removes and returns the last item
- pop(pos) removes and returns the item at a position

Left as an exercise
Try to adapt the methods from SLinkedList Class