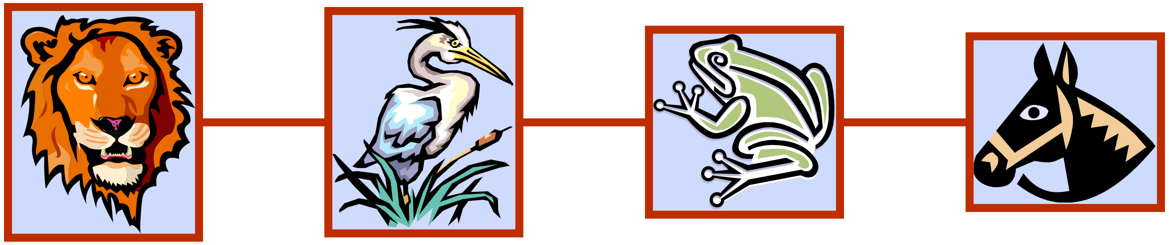
Linked Lists



Array vs Linked List

- Disadvantages of arrays as storage data structures
  - The length of a dynamic array might be longer than the actual number of elements that it stores.
  - Amortized bounds for operations may be unacceptable in real-time systems.
  - Insertions and deletions at interior positions of an array are expensive
  - Fixed size

Array vs Linked List

- Linked lists
  - More complex to code and manage
  - **Dynamic**
    - A linked list can easily grow and shrink in size
    - Data items (Nodes) are allocated in memory as needed
    - A dynamically linked sequence of Nodes
  - **Easy and fast insertions and deletions**
    - Only need to modify references: data items stay where they are

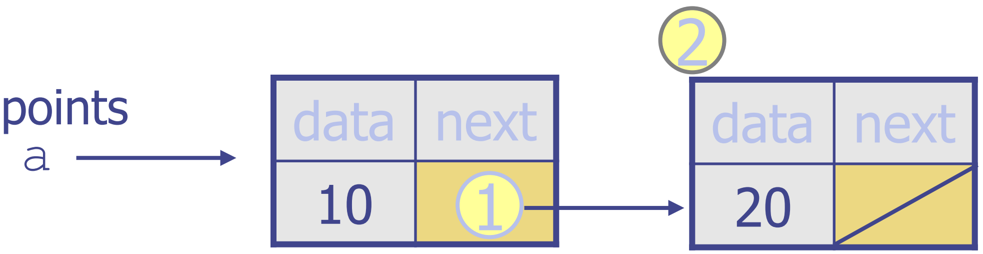
References vs. objects

variable = **value**

a *variable* (left side of =) is an arrow (the base of an arrow)

a *value* (right side of =) is an object (a box; what an arrow points at)

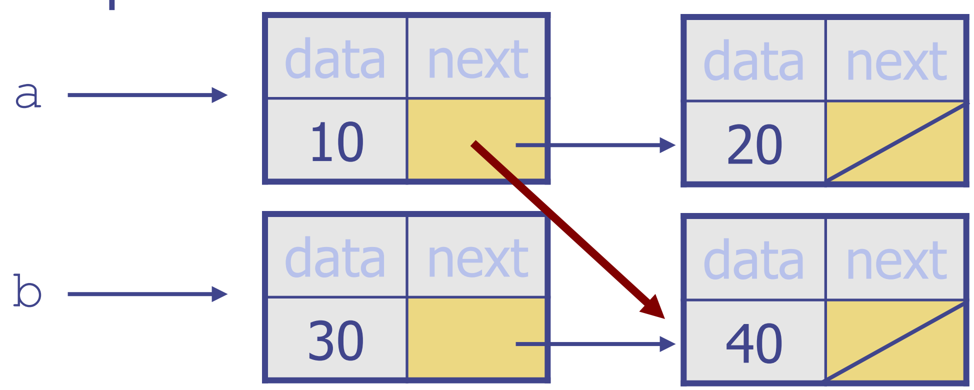
- For the list at right:
  - `a.next` = **value**
    means to adjust where 1 points
  - **variable** = `a.next`
    means to make **variable** point at 2



© 2013 Goodrich, Tamassia, Goldwasser

Reassigning references

- when you say:
  - `a.next = b.next;`
- you are saying:
  - "Make the *variable* `a.next` refer to the same *value* as `b.next`."
  - Or, "Make `a.next` point to the same place that `b.next` points."



© 2013 Goodrich, Tamassia, Goldwasser

The Many-Faced Data Structure

Linked Lists come in many forms

- Single-ended Singly Linked
- Double-Ended Singly Linked
- Circular (Not covered)
- Double-Ended Doubly Linked
- Positional List (not covered)

© 2013 Goodrich, Tamassia, Goldwasser

The Node Class for List Nodes

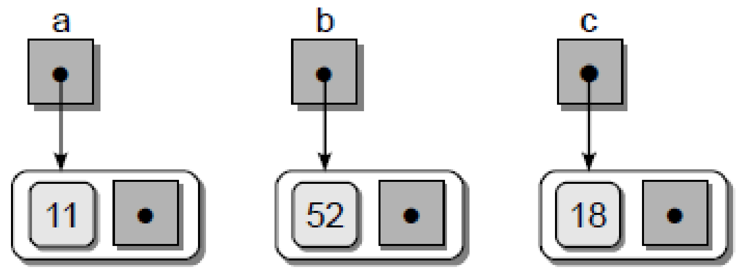
```
class Node:

    def __init__(self, element, next=None):
      # initialize node's fields

      self.data = element # reference to user's element
      self.next = next # reference to next node
```

Creating a LinkedList

a = Node(11)

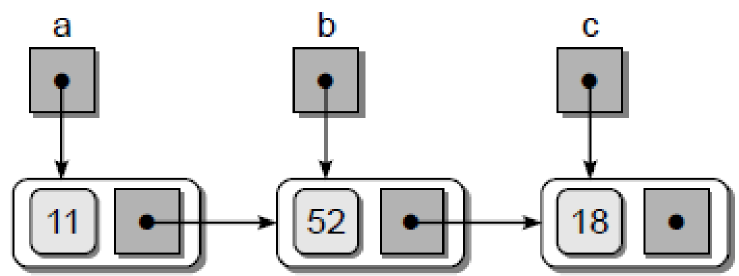
b = Node(52)

c = Node(18)



a.next = b

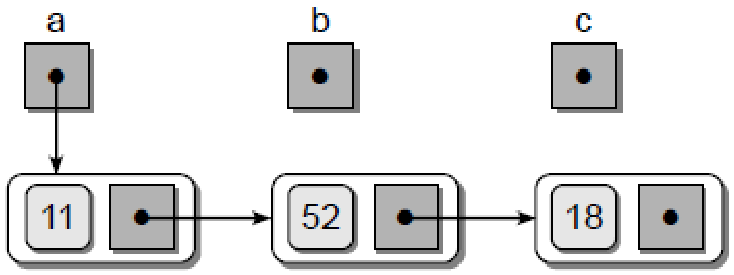
b.next = c



Creating a LinkedList

b = None

c = None



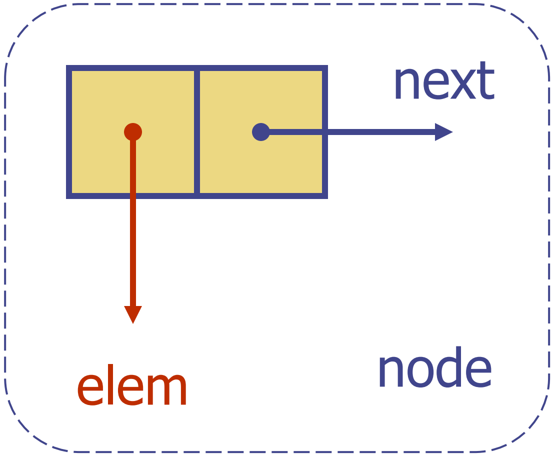

```
print( a.data )   # Output: 11
print( a.next.data ) # Output: 52
print( a.next.next.data ) # Output: 18
```

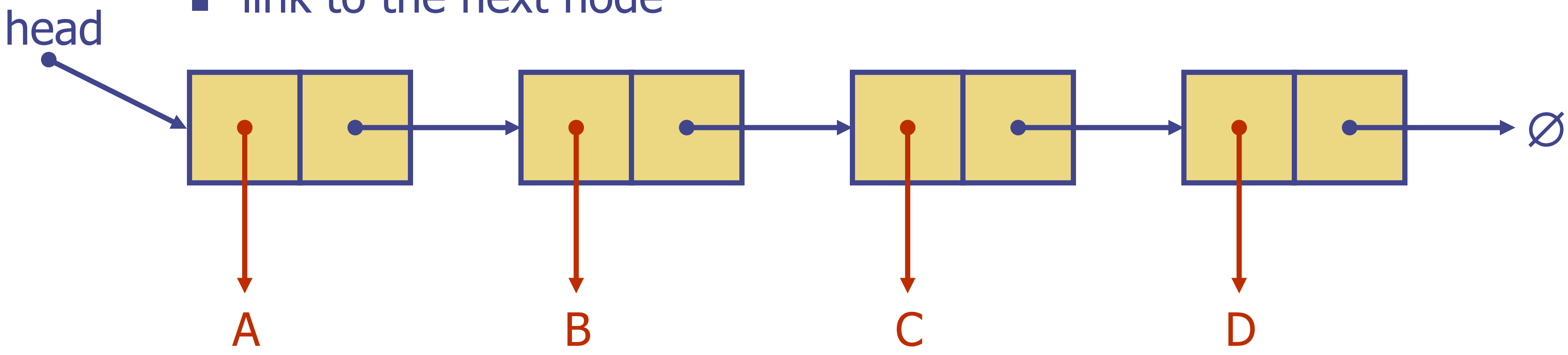
More terminology

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node



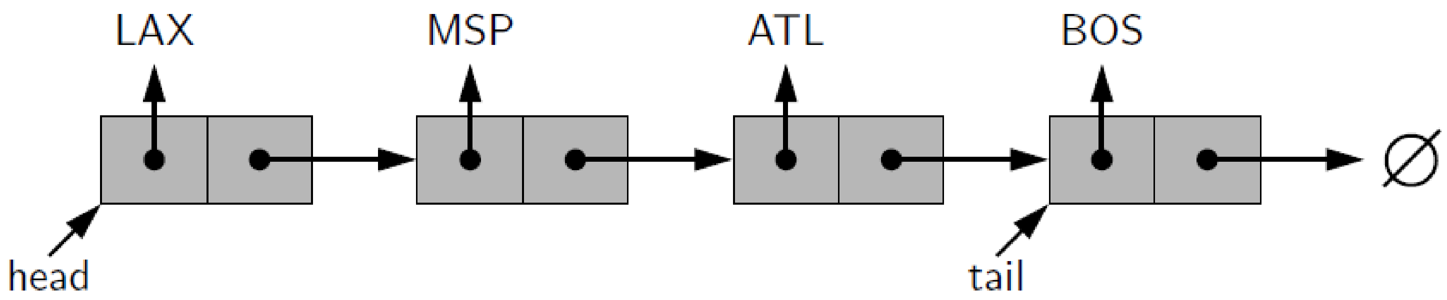


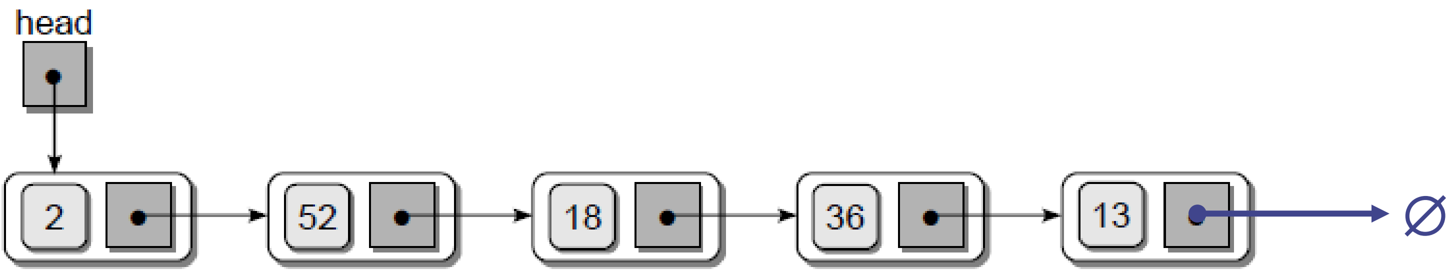
Python class of SLL

```
class Node:
  def __init__(self, element, next = None):   # initialize node's fields
    self._element = element                   # reference to user's element
    self._next = next                         # reference to next node
```

```
class Single_Linked_List:
  #---------------------------- Single Linked List methods ---------------
  def __init__(self):
    """Create an empty LinkedList."""
    self._head = None                         # reference to the head node
    self._size = 0                            # number of elements in the list
```

Other Single LinkedLists





Operations on an SLL

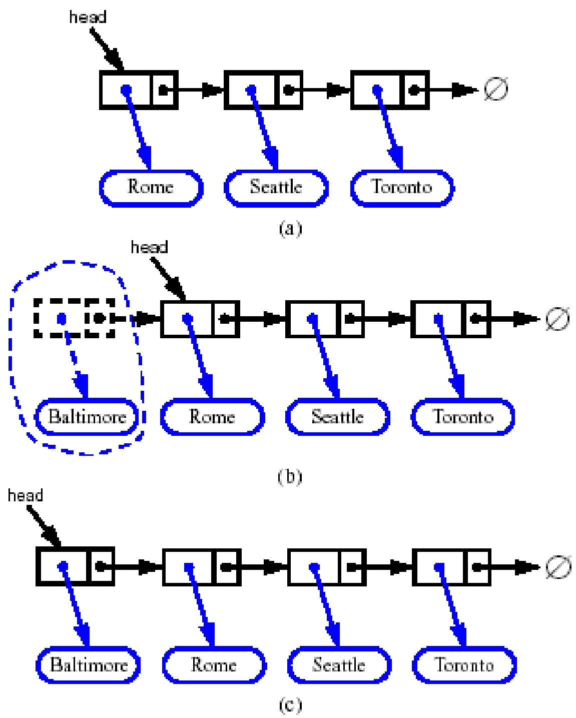
- Insertion
- Deletion
- Search or Iteration through the list to display items

Inserting/Removing an SLL Node

- Many ways to insert/remove a list node
  - As the new first element
  - As the new last element
  - Before a given node (specified by a *reference*)
  - After a given node
  - Before a given value
  - After a given value
- All are possible, but differ in difficulty

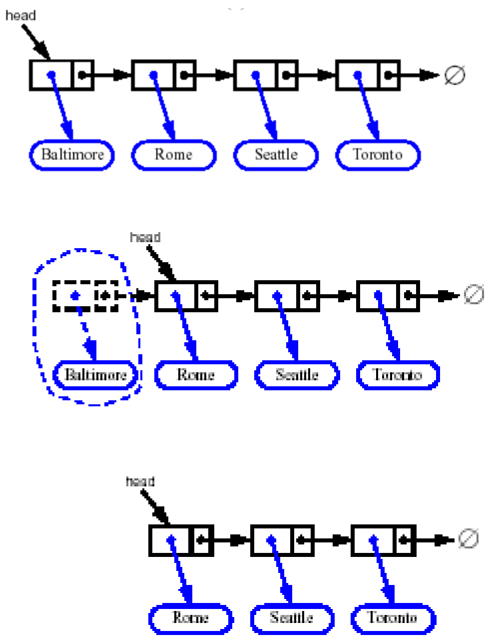
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node

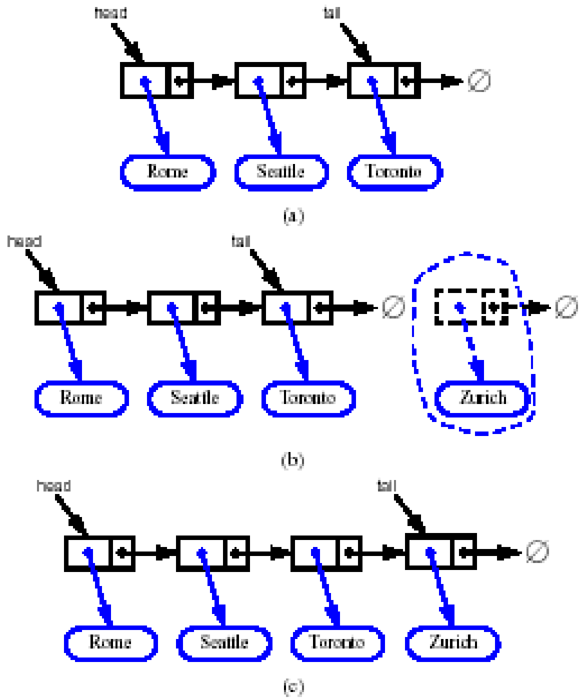



```
head = head.next
size -= 1
```

Inserting at the Tail

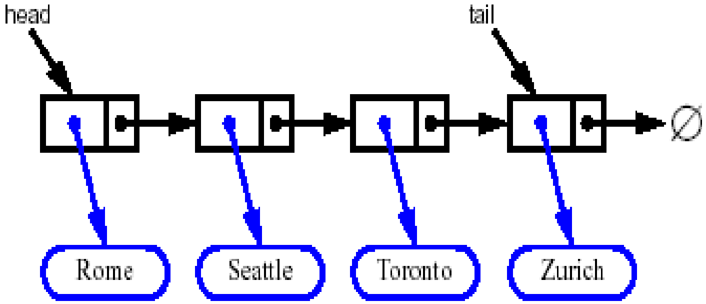
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

```
tail.next = Node("Zurich", next = None)
tail = tail.next
size += 1
```

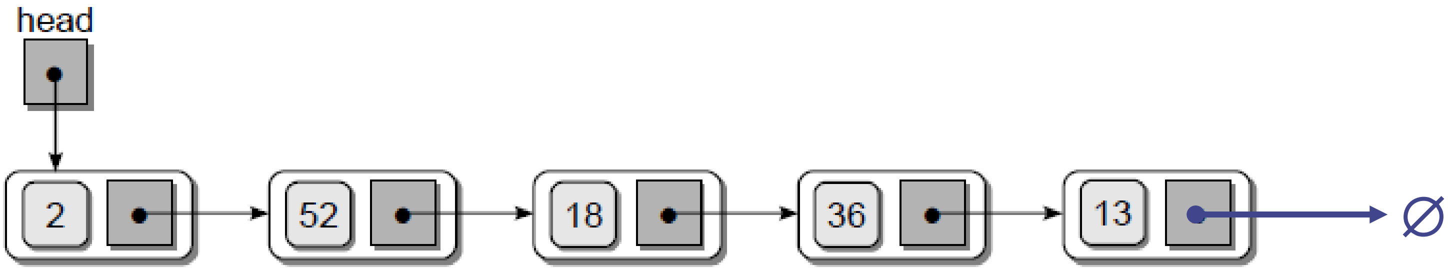


Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



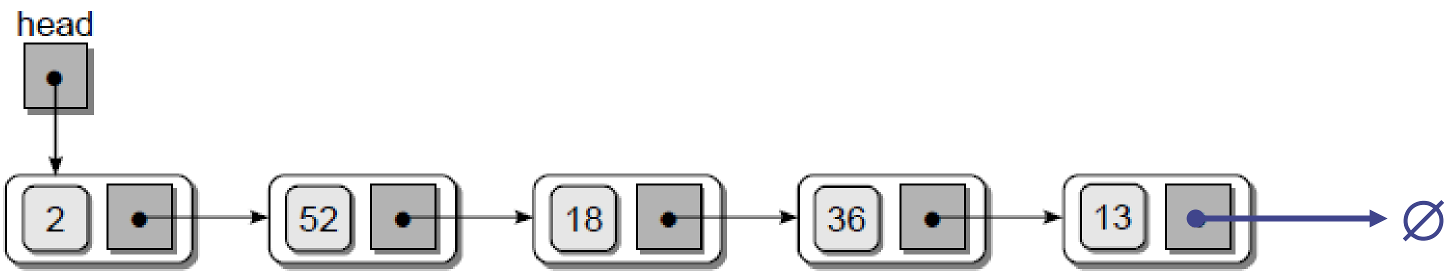
Traversing the Nodes




```
def traversal(self):
  while self._head is not None :
    print(self._head.data)
    self._head = self._head.next
```

Any issue here?

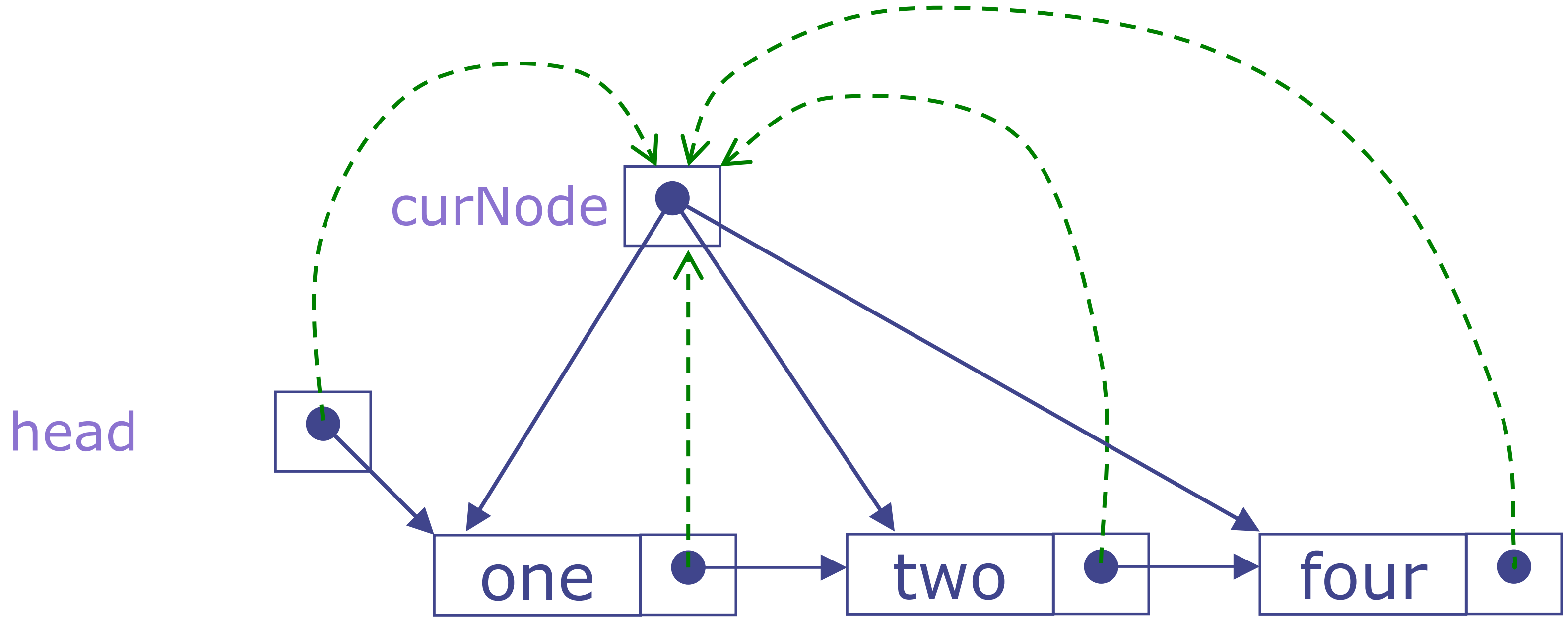
Traversing the Nodes




```
def traversal(self):
  curNode = self._head
  while curNode is not None :
     print(curNode.data)
     curNode = curNode.next
```

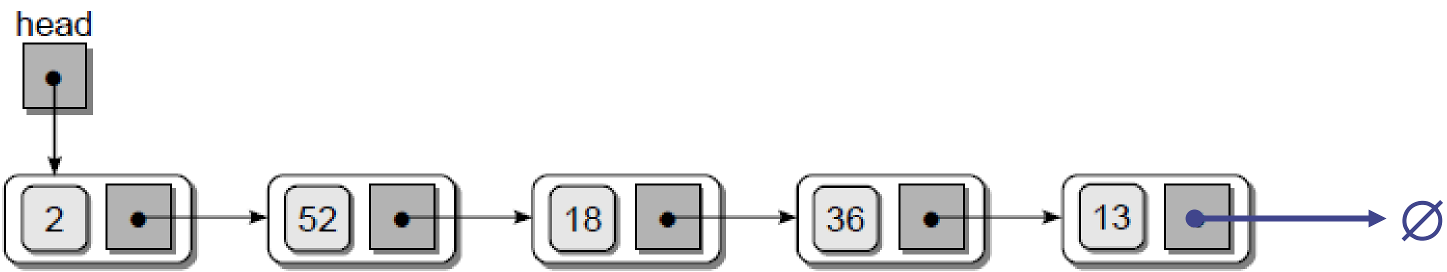
Traversing an SLL

Green arrow: assign a value to curNode



Traverse to the next node: curNode = curNode._next

Searching for a Node




```
def unorderedSearch(target):
   curNode = self._head
   while curNode is not None and curNode.data != target:
         curNode = curNode.next
   return curNode is not None
```

Linked List Efficiency

- Insertion and deletion at the beginning of the list are very fast

  O(1)

- Search, deletion, insertion require traversal

  O(n)

- Same number of comparisons as arrays

  But no shifting of items required after insertion or deletion

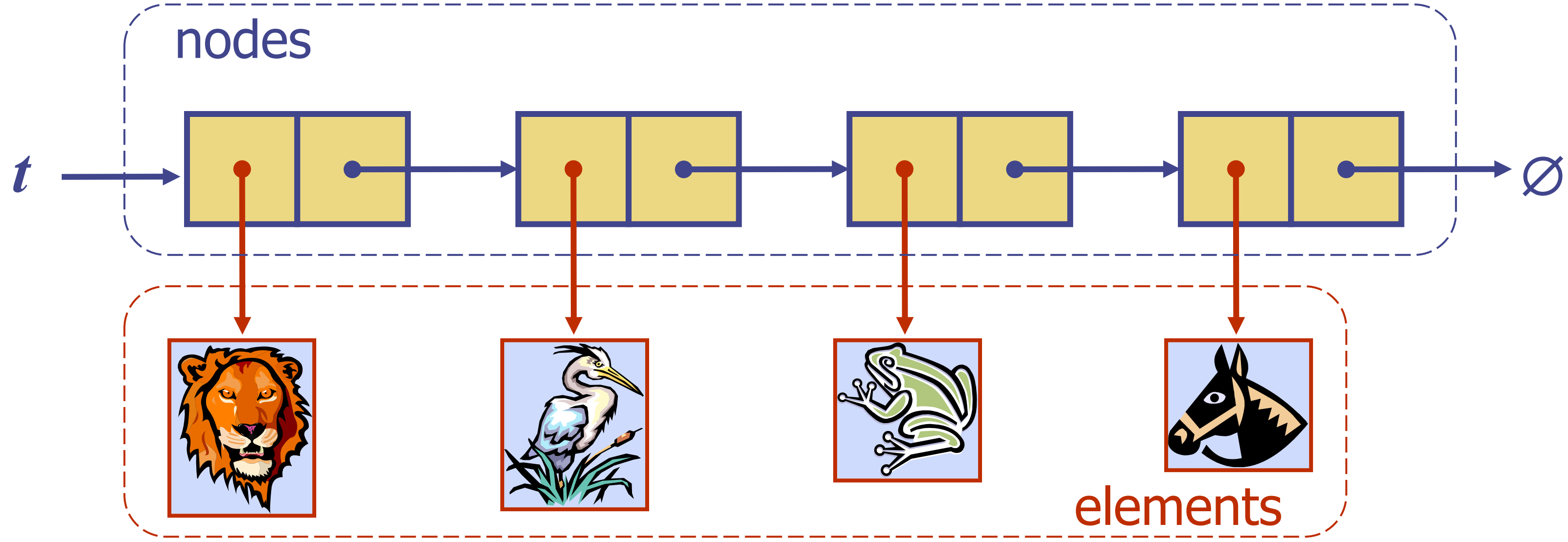
- Memory print strictly limited to usage and can shrink/expand (No need to pre-allocate with capacity)

Let’s Implement Single Linked List

- Download Single_linked_list_Students.py from Brightspace.
- Complete the insertAtFirst(e), deleteFirst() functions.
- Upload your solution to Gradescope

Stack as a Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



Let’s Implement a stack

- Among the functions that we implemented in Single_linked_list_Students.py, i.e insertAtFirst(e), deleteFirst() functions,
- which one can be applied to:
  - push(e)
  - pop()

Linked-List Stack in Python

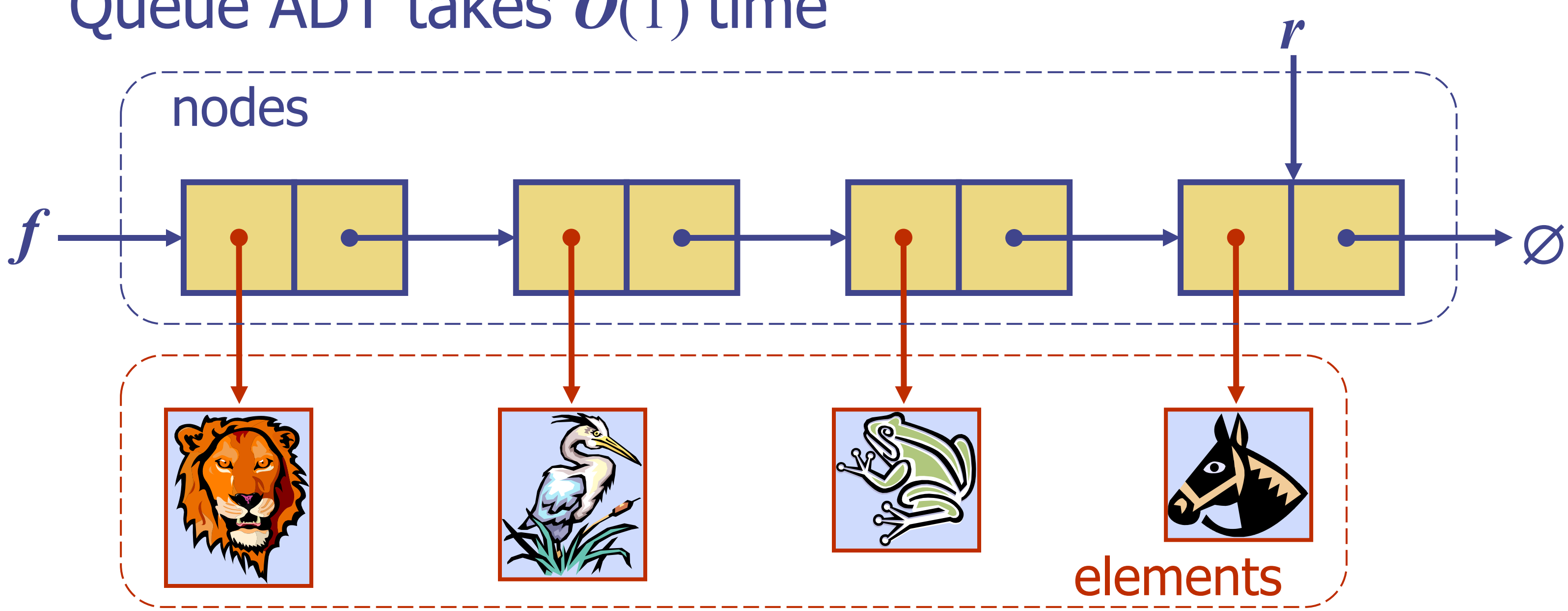
```python
class LinkedStack:
  """LIFO Stack implementation using a singly linked list for storage."""

  #------------------------- nested _Node class -------------------------
  class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    __slots__ = '_element', '_next'         # streamline memory usage

    def __init__(self, element, next):      # initialize node's fields
      self._element = element               # reference to user's element
      self._next = next                     # reference to next node

  #------------------------------- stack methods -------------------------------
  def __init__(self):
    """Create an empty stack."""
    self._head = None                       # reference to the head node
    self._size = 0                          # number of stack elements

  def __len__(self):
    """Return the number of elements in the stack."""
    return self._size

  def is_empty(self):
    """Return True if the stack is empty."""
    return self._size == 0

  def push(self, e):
    """Add element e to the top of the stack."""
    self._head = self._Node(e, self._head)  # create and link a new node
    self._size += 1

  def top(self):
    """Return (but do not remove) the element at the top of the stack.

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    return self._head._element              # top of stack is at head of list

  def pop(self):
    """Remove and return the element from the top of the stack (i.e., LIFO).

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    answer = self._head._element
    self._head = self._head._next           # bypass the former top node
    self._size -= 1
    return answer
```

Push: insertAtFirst

Pop: deleteFirst

Queue as a Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



Linked-List Queue in Python

```
class LinkedQueue:
  """FIFO queue implementation using a singly linked list for storage."""

  class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    (omitted here; identical to that of LinkedStack._Node)

  def __init__(self):
    """Create an empty queue."""
    self._head = None
    self._tail = None
    self._size = 0                        # number of queue elements

  def __len__(self):
    """Return the number of elements in the queue."""
    return self._size

  def is_empty(self):
    """Return True if the queue is empty."""
    return self._size == 0

  def first(self):
    """Return (but do not remove) the element at the front of the queue."""
    if self.is_empty():
      raise Empty('Queue is empty')
    return self._head._element            # front aligned with head of list

  def dequeue(self):
    """Remove and return the first element of the queue (i.e., FIFO).

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    answer = self._head._element
    self._head = self._head._next
    self._size -= 1
    if self.is_empty():                   # special case as queue is empty
      self._tail = None                   # removed head had been the tail
    return answer

  def enqueue(self, e):
    """Add an element to the back of queue."""
    newest = self._Node(e, None)          # node will be new tail node
    if self.is_empty():
      self._head = newest                 # special case: previously empty
    else:
      self._tail._next = newest
    self._tail = newest                   # update reference to tail node
    self._size += 1
```

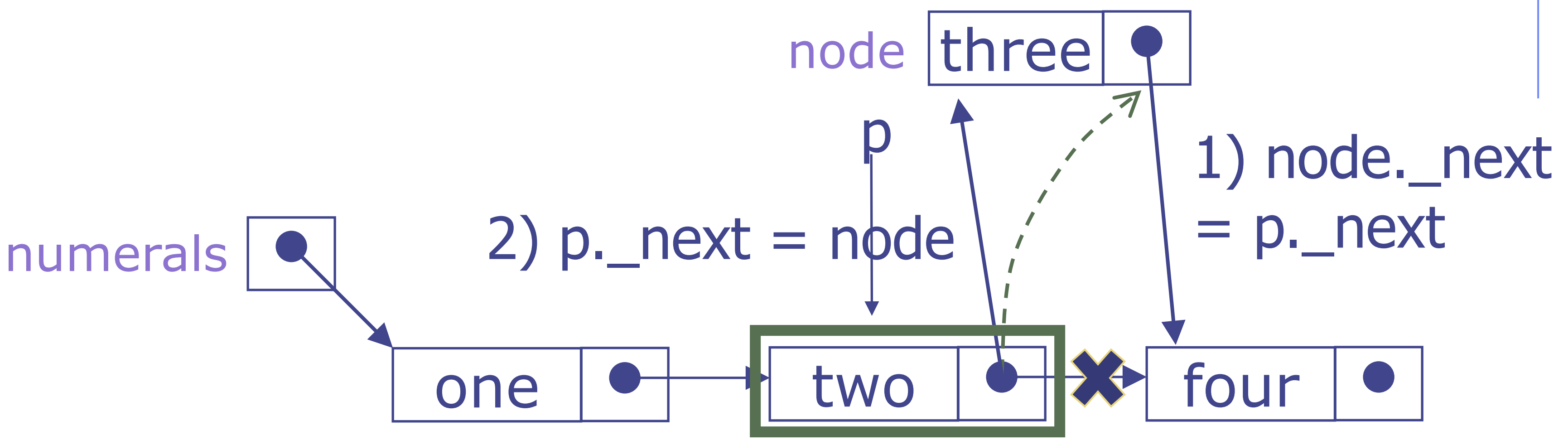
Queue as a Linked List

- Special Cases:

- When dequeue the last element, self._tail should be None too. That is, both self._head and self._tail should be None.
- When enqueue the first element, self._head should be newNode (line 45 in previous slide) too. That is, both self._head and self._tail should be newNode.
- To avoid the special cases, use header & trailer sentinels (we will see them in doubly-linked list soon)

© 2013 Goodrich, Tamassia, Goldwasser

Inserting After (More Operation of SLL)



Find the node you want to insert after

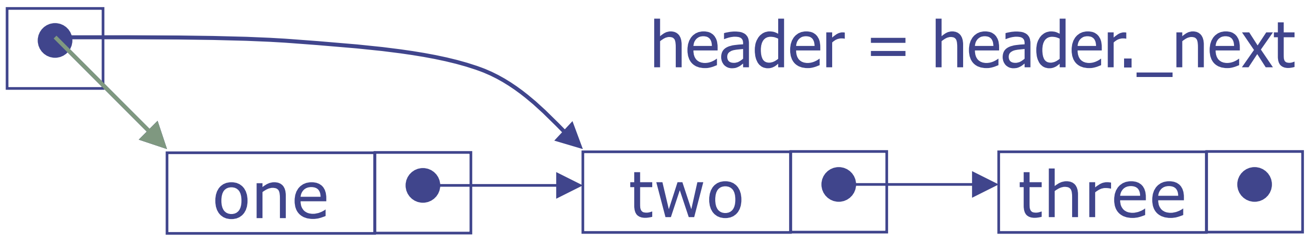
First, copy the link from the node that's already in the list

Then, change the link in the node that's already in the list

Deleting an Element From an SLL (More Operations of SLL)

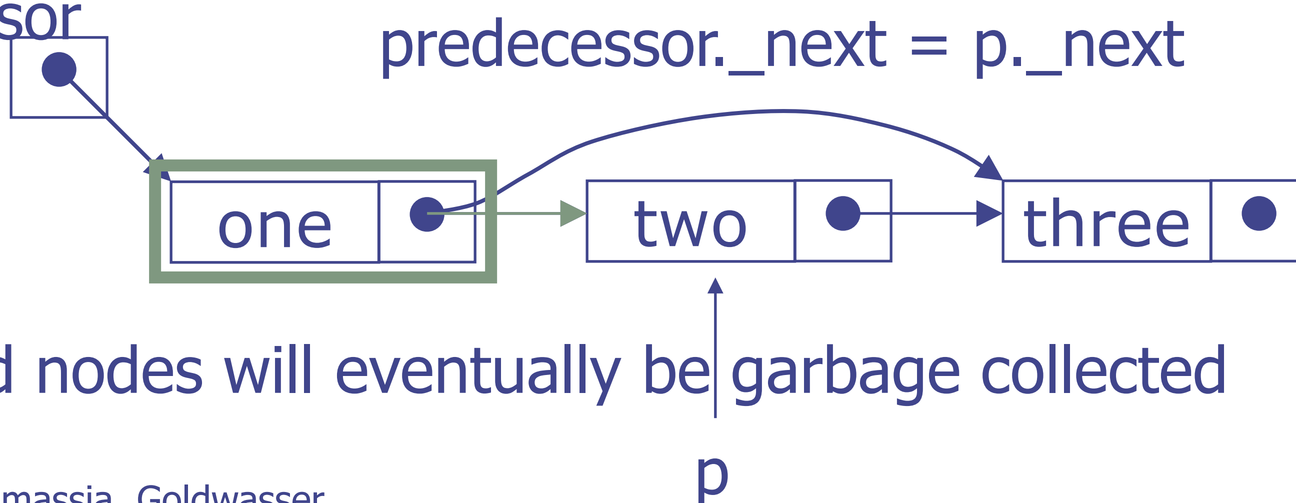
- To delete the first element, change the link in the header

numerals



- To delete some other element, change the link in its predecessor

numerals



- Deleted nodes will eventually be garbage collected

LinkedLists in Action

- *Visualization time*

  https://visualgo.net/en/list

The Many-Faced Data Structure

Linked Lists come in many forms

- Single-ended Singly Linked (head)
- Double-Ended Singly Linked (head + tail)
- Circular (Not covered)
- Double-Ended Doubly Linked (next ppt)
- Positional List (Not covered)

LinkedLists in Action

- Chapter 7 of text book and Exercises
- https://www.geeksforgeeks.org/data-structures/linked-list/
- https://leetcode.com/tag/linked-list/
- *Some slides have been prepared/taken from textbook's Website.*
- *Visualization time*

  https://visualgo.net/en/list

© 2013 Goodrich, Tamassia, Goldwasser