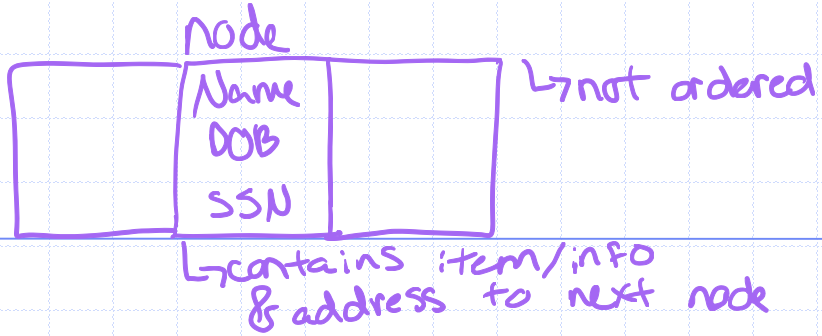
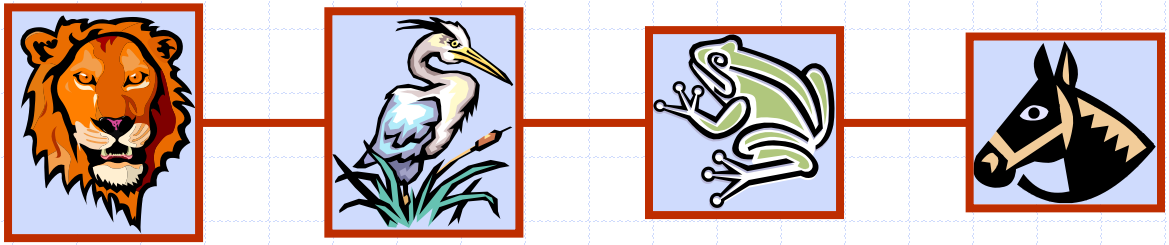


Linked Lists



Sareh Taebi
COP3410 – Florida Atlantic University

Array Sequences

- Python's list class is highly optimized, and often a great choice for storage.
- However, there are some notable disadvantages:

1. The length of a dynamic array might be longer than the number of elements that it stores.

↳using more memory than needed
↳"expensive" to access index element

2. Amortized bounds for operations may be unacceptable in real-time systems.(.append, .extend)

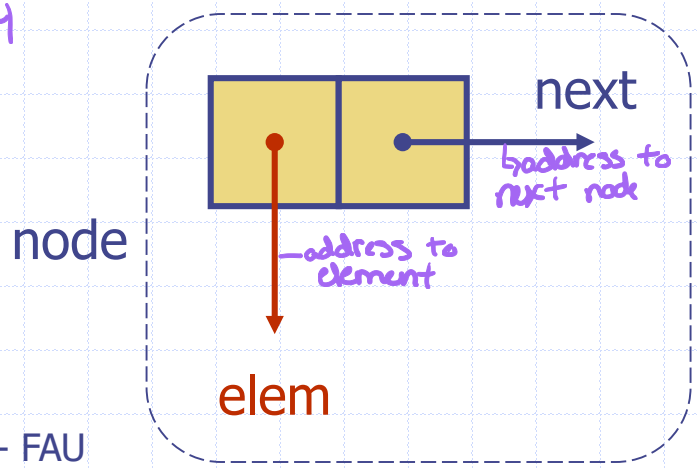
3. Insertions and deletions at interior positions of an array are expensive.

Linked-List (I)

- Linked list makes it easy to add, remove or delete elements from anywhere in the sequence.
- It does however need more memory space!
- Linked list consists of nodes.
- Nodes reference the item stored and the item that is next in line.
- Nodes are not stored in memory sequentially.

↳optimize memory

hexadecimals
0-15
$F \times 16^2$ 16^1 16^0
A F 0 C



Linked List (II)

- Elements of a linked list do not use the indexing system.
- We cannot tell if an element is 1^{st}, 2^{nd} or 5^{th} .
- Linked lists are considered as a class of **non-contiguous** data structures.
  - They occupy various locations in memory.
  - So, we need to browse the entire list to find a specific item in a linked list.

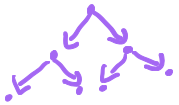
2 Types of Advanced Data Structures

Contiguous

- elements are stored continuously, in order
  - Stack
  - Queue
  - Built-ins
    - list
    - tuple

Non-contiguous

- Social media
  - Tree

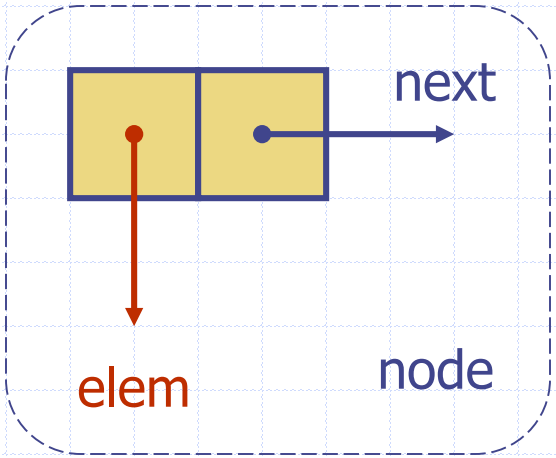


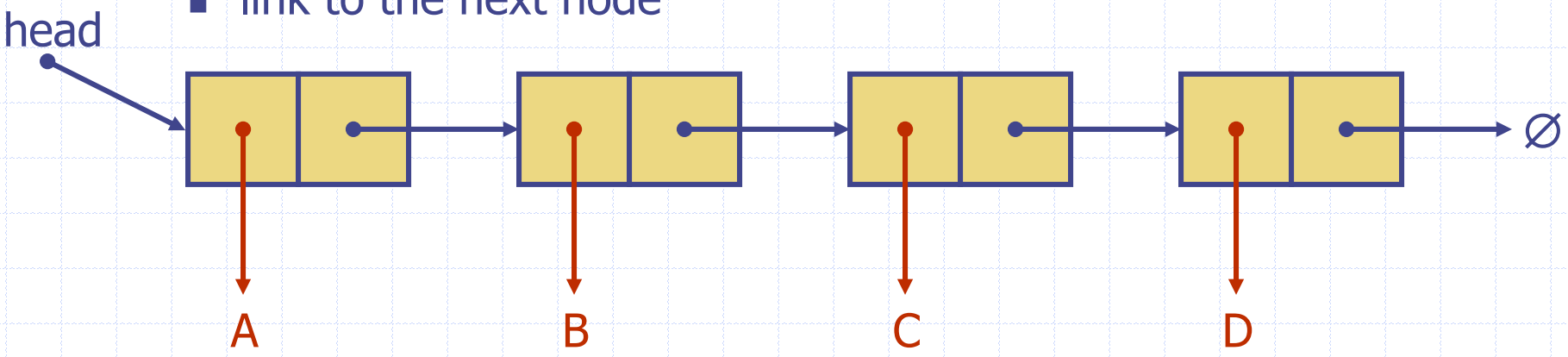
- graph
  - internet

- What actions can a Link List support?
  - create a node @ head or tail
  - insert data
  - removal of node
  - traversing link list

Singly Linked List

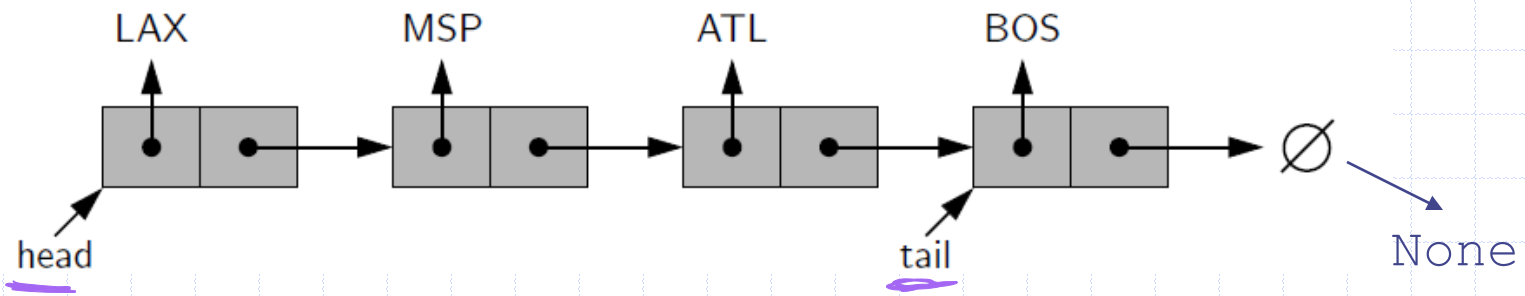
- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node



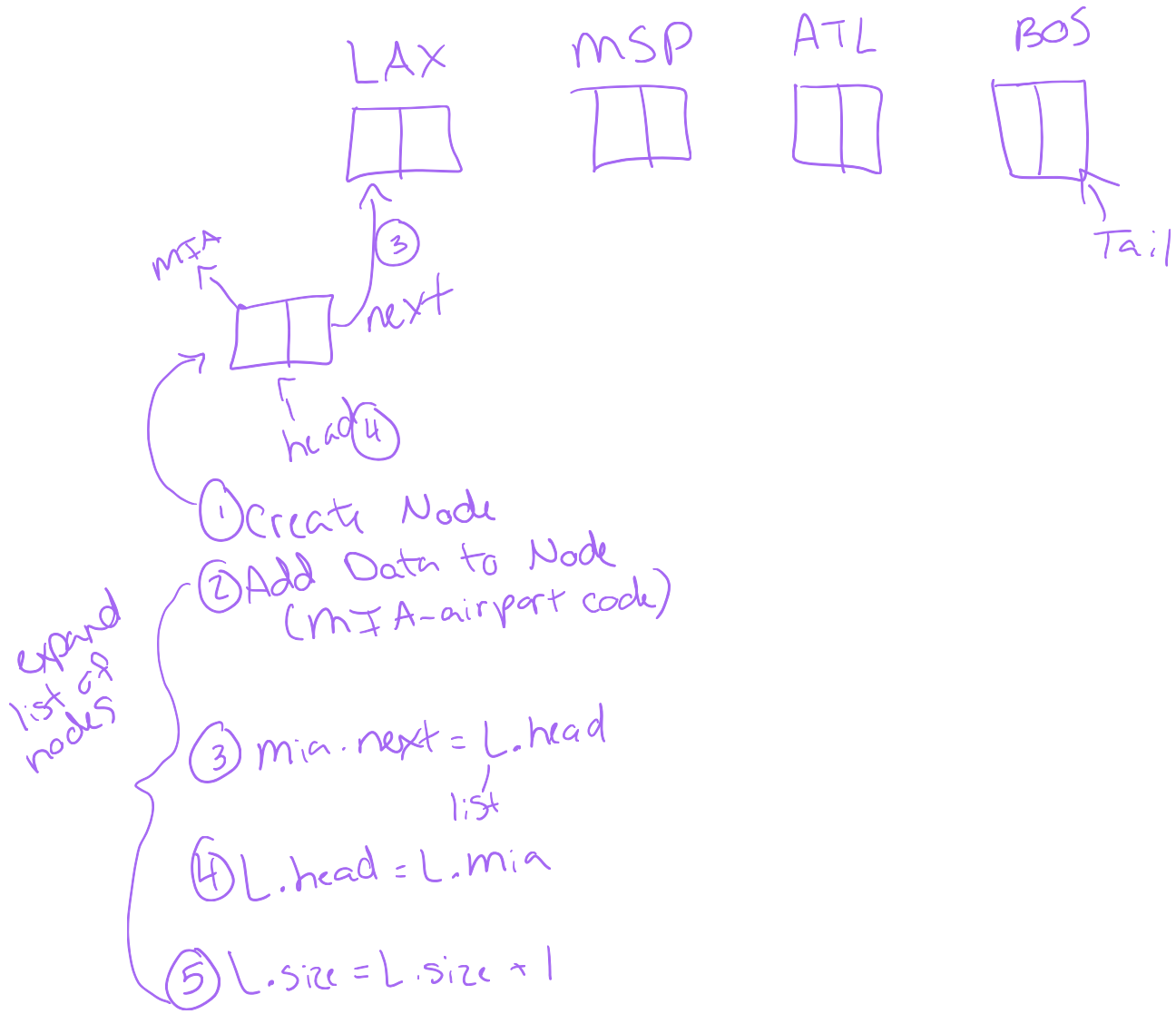


Linked list Operations

Use more memory than array



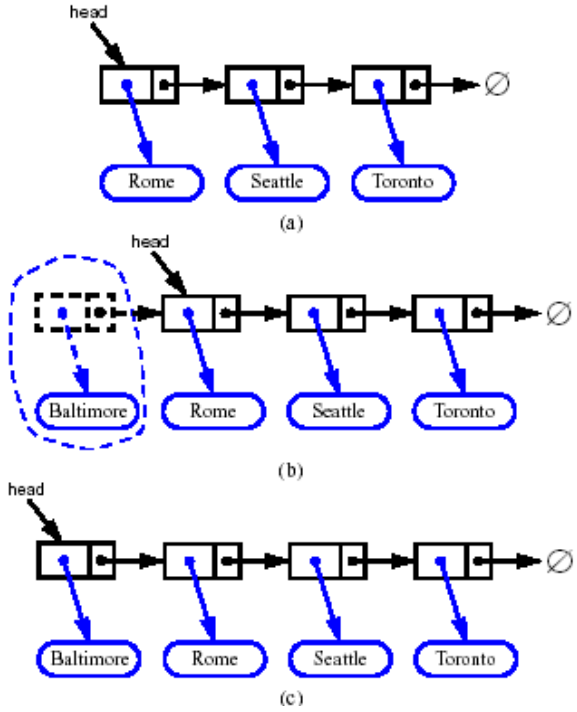
- Creation of a node
- Insertion of a node to the begin, middle or end.
- Deletion of a node
- Traversing the linked list: Begin at head and use each reference to reach the end (Indicated by None)


LAX
MSP
ATL
BOS
Tail
MIA
3
next
head
4
1 Create Node
2 Add Data to Node
(MIA-airport code)
expand list of nodes
3 mia.next = L.head
list
4 L.head = L.mia
5 L.size = L.size + 1

Stack: → LIFO
↳ treat stack like link list
↳ add on stack → add to head
↳ use Node for stack implementation
↳ not array

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



e = new element (MIA)

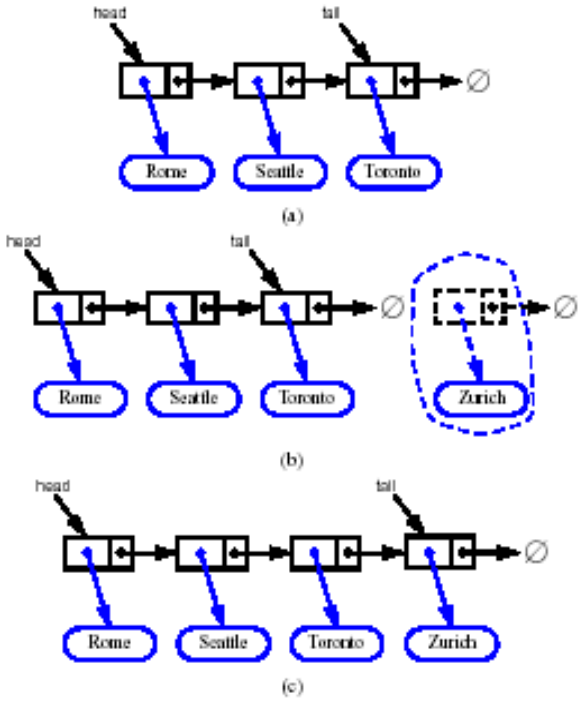
Algorithm add_first(L, e):

newest = Node(e) {create new node instance storing reference to element e}
newest.next = L.head {set new node's next to reference the old head node}
L.head = newest {set variable head to reference the new node}
L.size = L.size + 1 {increment the node count}

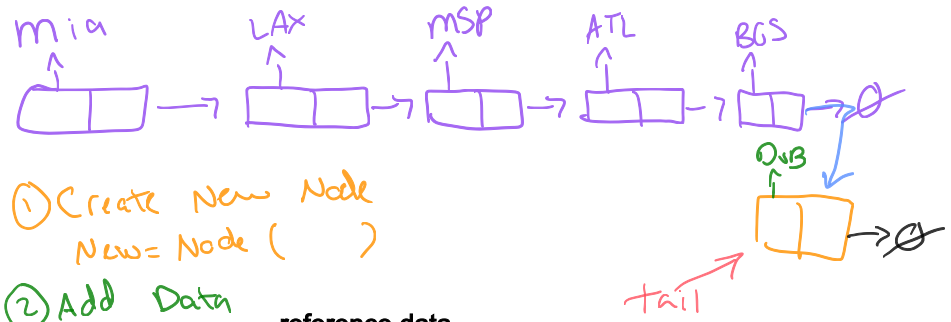
↳ keep track of size
↳ or have to traverse whole list to count

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



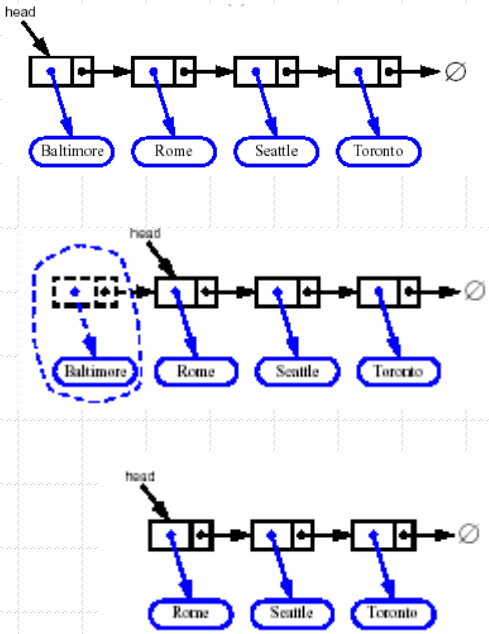
Insert @ tail



1. Create New Node
   New = Node ( )
2. Add Data
   New = DUB reference data
3. tail.next = New
   Connect current tail to New
4. New.next = None
5. Move tail (optional)
   L.tail = new
6. L.size = L.size + 1
   add to size

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

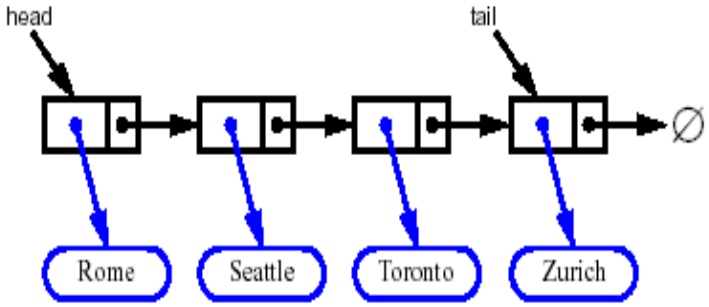


O(1) complexity

L.head = L.head.next

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



Not efficient for single linked list
have to start at beginning and go through all until end

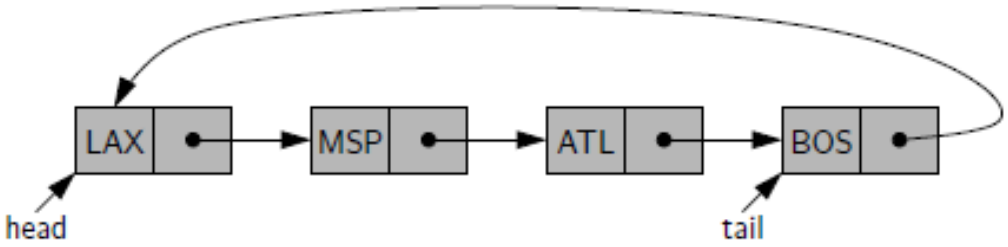
The Node Class

```
class Node:
    '''Lightweight class for storing a singly linked node.'''

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

    def getElement(self):                      # Accessor methods
        return  self.element

    def getNext(self):                       # Accessor methods
        return  self.next

    def setElement(self,new):                  # Modifier methods
        self.element  =  new

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

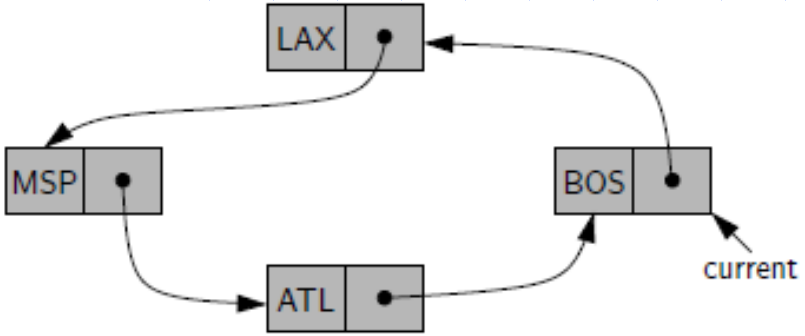
Refer to the python code

Circularly Linked Lists

- Used for data sets that are cyclic : players taking turn in a game
- No begin or end node
- The 'current' identifier indicates the reference to a designated node.
- `current = current.next` helps advance through nodes
  - head/tail : depends on whose turn it is

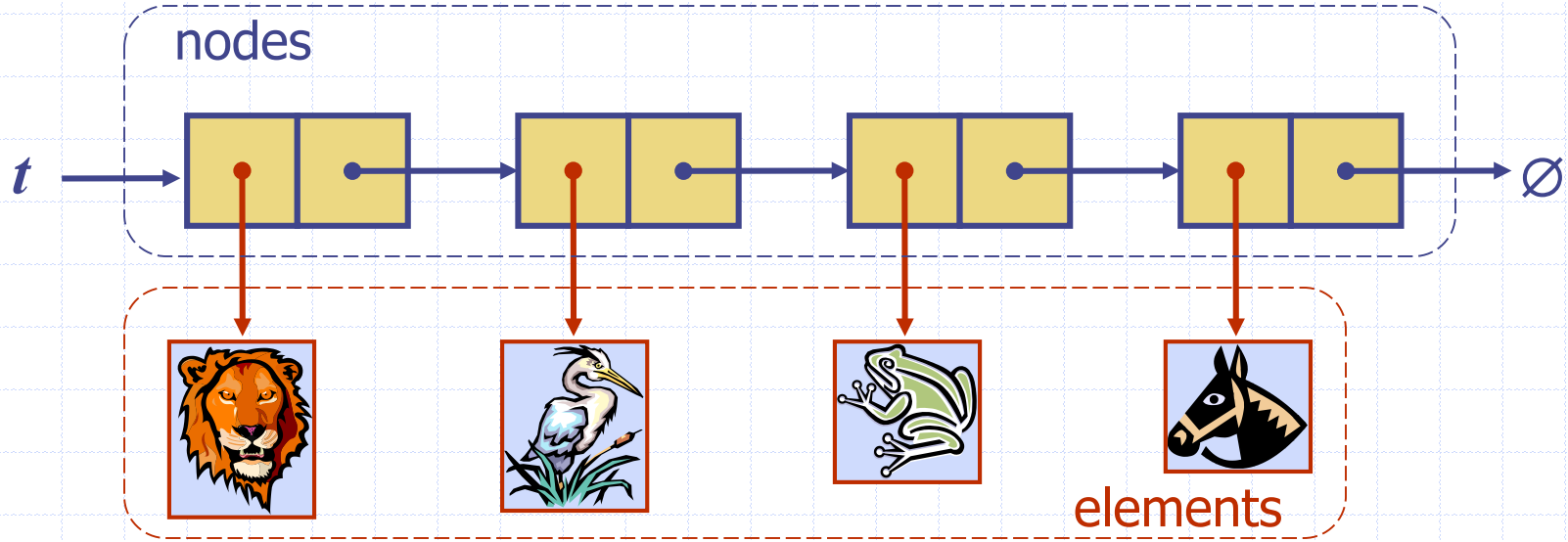


- how to tell if circular?
  - circular : no none at end



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



Queue FIFO easier to remove head than tail

Linked-List Stack in Python

```
1  class LinkedStack:
2    """LIFO Stack implementation using a singly linked list for storage."""
3
4    #---------------------------- nested _Node class ----------------------------
5    class _Node:
6      """Lightweight, nonpublic class for storing a singly linked node."""
7      __slots__ = '_element', '_next'         # streamline memory usage
8
9      def __init__(self, element, next):      # initialize node's fields
10       self._element = element               # reference to user's element
11       self._next = next                     # reference to next node
12
13   #------------------------------ stack methods ------------------------------
14   def __init__(self):
15     """Create an empty stack."""
16     self._head = None                       # reference to the head node
17     self._size = 0                          # number of stack elements
18
19   def __len__(self):
20     """Return the number of elements in the stack."""
21     return self._size
22
```

```
23   def is_empty(self):
24     """Return True if the stack is empty."""
25     return self._size == 0
26
27   def push(self, e):
28     """Add element e to the top of the stack."""
29     self._head = self._Node(e, self._head)     # create and link a new node
30     self._size += 1
31
32   def top(self):
33     """Return (but do not remove) the element at the top of the stack.
34
35     Raise Empty exception if the stack is empty.
36     """
37     if self.is_empty():
38       raise Empty('Stack is empty')
39     return self._head._element                 # top of stack is at head of list
```

```
40   def pop(self):
41     """Remove and return the element from the top of the stack (i.e., LIFO).
42
43     Raise Empty exception if the stack is empty.
44     """
45     if self.is_empty():
46       raise Empty('Stack is empty')
47     answer = self._head._element
48     self._head = self._head._next              # bypass the former top node
49     self._size -= 1
50     return answer
```

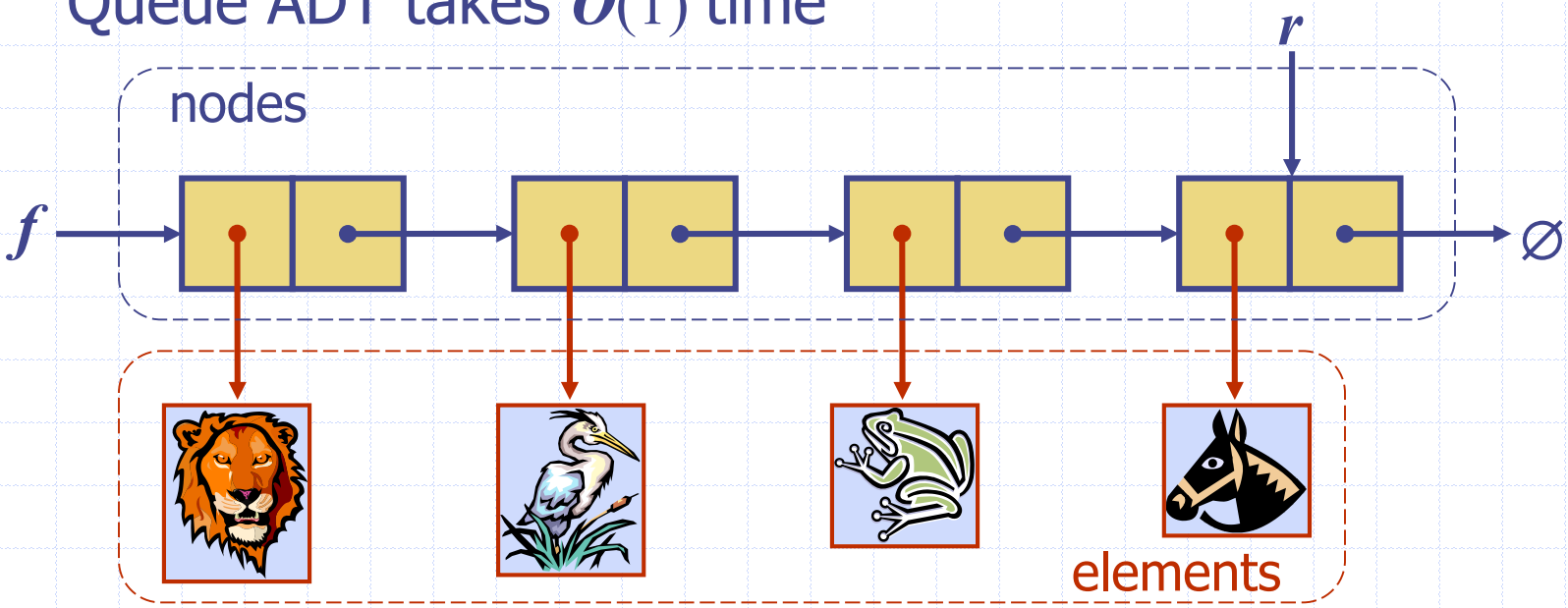
Linked Stack Implementation

- All methods complete in worst case O(1) time.
- Huge improvement over array-based stack ADT.

Operation	Running Time
S.push(e)	$O(1)$
S.pop( )	$O(1)$
S.top( )	$O(1)$
len(S)	$O(1)$
S.is_empty( )	$O(1)$

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

```
1  class LinkedQueue:
2    """FIFO queue implementation using a singly linked list for storage."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a singly linked node."""
6      (omitted here; identical to that of LinkedStack._Node)
7
8    def __init__(self):
9      """Create an empty queue."""
10     self._head = None
11     self._tail = None
12     self._size = 0                          # number of queue elements
13
14   def __len__(self):
15     """Return the number of elements in the queue."""
16     return self._size
17
18   def is_empty(self):
19     """Return True if the queue is empty."""
20     return self._size == 0
21
22   def first(self):
23     """Return (but do not remove) the element at the front of the queue."""
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._head._element              # front aligned with head of list
```

```
27   def dequeue(self):
28     """Remove and return the first element of the queue (i.e., FIFO).
29
30     Raise Empty exception if the queue is empty.
31     """
32     if self.is_empty():
33       raise Empty('Queue is empty')
34     answer = self._head._element
35     self._head = self._head._next
36     self._size -= 1
37     if self.is_empty():                     # special case as queue is empty
38       self._tail = None                     # removed head had been the tail
39     return answer
40
41   def enqueue(self, e):
42     """Add an element to the back of queue."""
43     newest = self._Node(e, None)            # node will be new tail node
44     if self.is_empty():
45       self._head = newest                   # special case: previously empty
46     else:
47       self._tail._next = newest
48     self._tail = newest                     # update reference to tail node
49     self._size += 1
```