CS 260: Foundations Of Computer Science

Class 18 – October 3, 2012

1

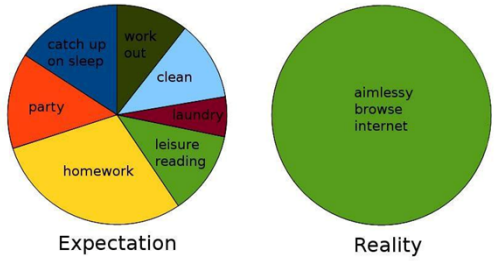
Thought for the Day

Be thankful for problems.

If they were less difficult, someone with less ability might have your job.

2

Weekend in college



Today's Agenda

- Chapter 5 – Finish Revisiting older ADTs
- Read chapter 6 for today

5

Set ADT Revisited

- We examined the worst come run-times in the previous chapter.
  - The *contains* operation required linear time due to the linear search.
  - The quadratic times of some operations were due to the use of the linear *contains* operation.

Set ADT: Run-Times

List Operation	Worst Case
s = Set()	O(1)
len(s)	O(1)
x in s	O(n)
s.add(x)	O(n)
s.isSubsetOf(t)	$O(n^2)$
s == t	$O(n^2)$
s.union(t)	$O(n^2)$
traversal	O(n)

If we could improve the speed of **these** operations…

Set ADT: Sorted List

- Can the efficiency of the set operations be improved if we used a sorted list?
- What changes would be necessary?

Set Class: Sorted List

binaryset.py

```
class Set :
  def __init__( self ):
    self._theElements = list()

  def __len__( self ):
    return len( self._theElements )

  def __contains__( self, element ):
    ndx = self._findPosition( element )
    return ndx < len( self ) and\
           self._theElements[ndx] == element
# ...
```


Set Class: Sorted List

binaryset.py

```
class Set :
# ...
  def add( self, element ):
    if element not in self :
      ndx = self._findPosition( element )
      self._theElements.insert( ndx, element )


  def remove( self, element ):
    assert element in self,
           "The element must be in the set."
    ndx = self._findPosition( element )
    return self._theElements.pop( ndx )
```


Set Class: Sorted List

binaryset.py

```
class Set :
# ...
  def isSubsetOf( self, setB ):
    for element in self :
      if element not in setB :
        return False
    return True
```


Comparing Implementations

Operation	Linear Set	Binary Set
s = Set()	O(1)	O(1)
len(s)	O(1)	O(1)
x in s	O(n)	O(log n)
s.add(x)	O(n)	O(n)
s.isSubsetOf(t)	$O(n^2)$	O(n log n)
s == t	$O(n^2)$	
s.union(t)	$O(n^2)$	
traversal	O(n)	O(n)

New Set Equals

- If we use the original `isSubsetOf()`, the result is a worst case time of O(n log n). But ...

```
class Set :
# ...
  def __eq__( self, setB ):
    if len( self ) != len( setB ) :
      return False
    else :
      return self.isSubsetOf( setB )
```

New Set Equals

- A more efficient implementation is possible.

```
class Set :
# ...
  def __eq__( self, setB ):
    if len( self ) != len( setB ) :
      return False
    else :
      for i in range( len(self) ) :
        if self._theElements[i] != setB._theElements[i] :
          return False
      return True
```


New Set Union

- The efficiency of the set union operation can also be improved.
- Set union using two sorted lists is very similar to the problem of merging two sorted lists.
  - The new list only contains unique elements.
  - If both lists contains a given element, only one instance is placed in the new list.

Set Class: Sorted List

binaryset.py

```
class Set :
# ...
  def union( self, setB ):
    newSet = Set()
    a = 0
    b = 0
    while a < len( self ) and b < len( setB ) :
      valueA = self._theElements[a]
      valueB = setB._theElements[b]
      if valueA < valueB :
        newSet._theElements.append( valueA )
        a += 1
      elif valueA > valueB :
        newSet._theElements.append( valueB )
        b += 1
      else :
        newSet._theElements.append( valueA )
        a += 1
        b += 1
```


Set Class: Sorted List

binaryset.py

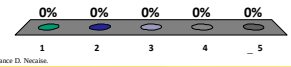
```
class Set :
# ...
  def union( self, setB ):
  # ...
    while a < len( self ) :
      newSet._theElements.append(self._theElements[a])
      a += 1

    while b < len( setB ) :
      newSet._theElements.append(setB._theElements[b])
      b += 1

    return newSet
```


Comparing Implementations

Operation	Linear Set	Binary Set
s = Set()	O(1)	O(1)
len(s)	O(1)	O(1)
x in s	O(n)	**O(log n)**
s.add(x)	O(n)	O(n)
s.isSubsetOf(t)	$O(n^2)$	**O(n)**
s == t	$O(n^2)$	**O(n)**
s.union(t)	$O(n^2)$	**O(n)**
traversal	O(n)	O(n)

On a scale of 1-5, how comfortable are you with this material so far?

1. Comfortable as LSU trying to cross the 50-yard line!
2. …
3. …
4. …
5. Comfortable as Alabama watching LSU try!



About Sorting…

- Check out this website that visualizes sorts:

  Video: "Sorting Out Sorts"

  http://video.google.com/videoplay?docid=-4110947752111188923

  Interactive visualization of sorts:

  http://visualsort.appspot.com/

Linked Structures

Chapter 6

Review

- Arrays
  - Basic sequence container
  - Provides easy and direct element access.
  - Supported at the hardware level.
  - Limited functionality.
  - Fixed size.

Review

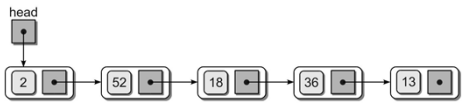
- Python list
  - Extends array functionality.
  - Provides a larger set of operations.
  - Can automatically adjust size as needed.
  - Change in size requires allocation of new array.
  - Insertions and deletions require items to be shifted.

Introduction

- In this chapter, we introduce the linked list data structure.
  - Can be used to store a collection in linear order.
  - Improves on the construction and management of an array and list.
  - Requires smaller memory allocations.
  - No element shifts for insertions and deletions.
  - Eliminates constant time direct element access.

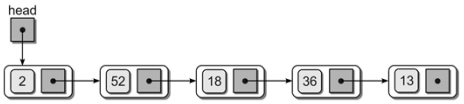
Linked Structure

- Constructed using a collection of objects called **nodes**.
- Each node contains data and at least one reference or **link** to another node.
- **Linked list** – a linked structure in which the nodes are linked together in linear order.



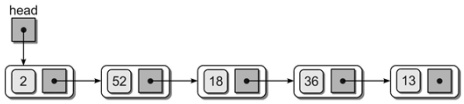
Linked List

- Terms:
  - **head** – first node in the list.
  - **tail** – last node in the list; link field has a null reference.



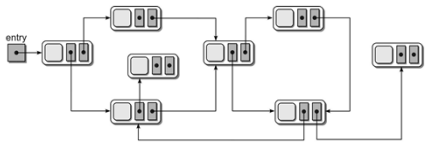
Linked List

- Most nodes have no name; they are referenced via the link of the preceding node.
- **head reference** – the first node must be named or referenced by an external variable.
  - Provides an entry point into the linked list.
  - An empty list is indicated by a null head reference.



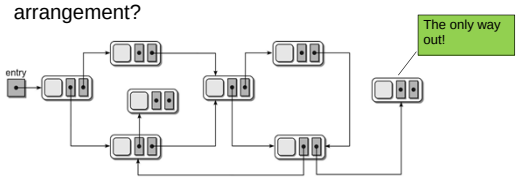
Linked List

- Linked list are built using fundamental components provided by the language:
  - reference variables
  - objects
- Many different configurations are possible:



Linked List – What could you do?

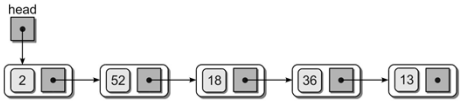
- What kind of data would make sense for this kind of arrangement?



- How about a game: "You are in a maze of twisty little tunnels, all alike."

Singly Linked List

- A linked list in which
  - each node contains a single link field and
  - allows for a complete linear order traversal from front to back.
- Several common operations can be performed.



Node Definition

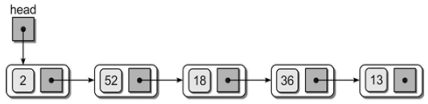
- The nodes are constructed from a simple storage class:

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

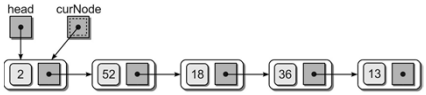
"Data" can be ***anything***, from a character or number to a "Student" object, etc.

Traversing the Nodes

- We can traverse the nodes using a temporary external reference variable.



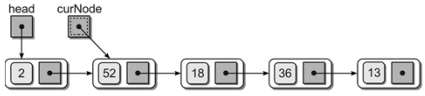
- Initialize a temporary reference to the head node.



- Visit the node.

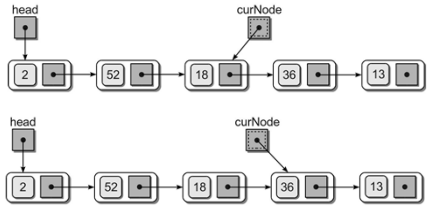
Traversing the Nodes

- Advance the temporary reference to the next node using the link field and visit that node.



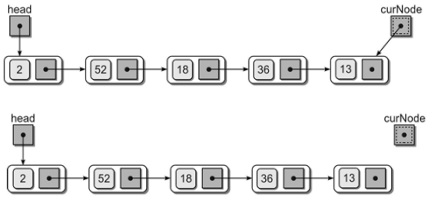
Traversing the Nodes

- Repeat the process until the reference falls off the end of the list.



Traversing the Nodes

- Repeat the process until the reference falls off the end of the list.



Traversal Code

- Given the head reference, we can traverse the nodes.

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


Traversing a List

- Many, many list operations are based on traversing a list.

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

This "print" step can be replaced by any number of things.

Traversing a List

- Many, many list operations are based on traversing a list.

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

You should memorize this loop. It will come back **so many** times!

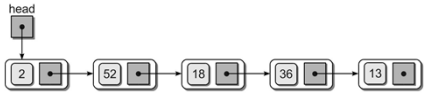
Searching

- We can perform a linear search to determine if the list contains a specific data item.

```
def unorderedSearch( head, target ):
  curNode = head
  while curNode is not None and
        curNode.data != target :
    curNode= curNode.next

  return curNode is not None
```

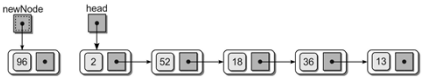

Prepending Nodes

- When working with an unsorted linked list, new values can be inserted at any point.
- We can prepend new items with little effort.
- **Example:** add value 96 to the sample list.

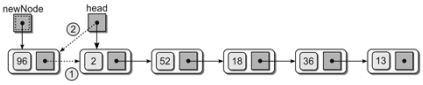


Prepending Nodes

- Create a new node for the new item.

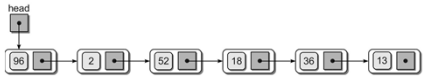


- Connect the new node to the list.



Prepending Nodes

- The resulting list.

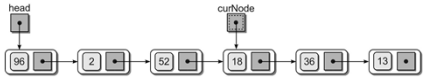


- Python code

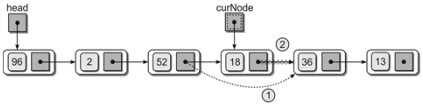
```
# Given the head reference and the new item.
newNode = ListNode( newItem )
newNode.next = head
head = newNode
```


Removing Nodes

- An item can be removed from a linked list by removing or unlinking the node containing the item.
  - Find the node containing the item.

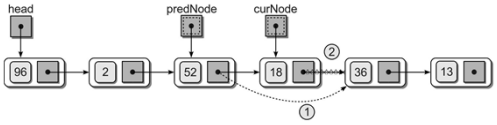


  - Unlink it from the list.

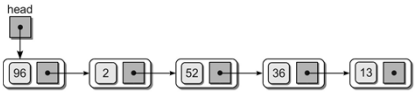


Removing Nodes

- Removing a node from the middle of the list requires a second external reference.

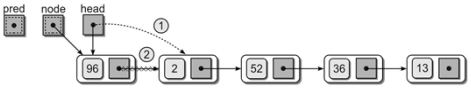


- Resulting list.



Removing Nodes

- Removing the first node is a special case.
- The head reference must be reposition to reference the next node in the list.



Removing Nodes

- Given the head reference, we can remove a target from a linked list.

```
predNode = None
curNode = head
while curNode is not None and curNode.data != target :
  predNode = curNode
  curNode = curNode.next

if curNode is not None :
  if curNode is head :
    head = curNode.next
  else :
    predNode.next = curNode.next
```