COMPSCI 105 S1 2017
Principles of Computer Science

17 Linked List(1)

Agenda & Readings

- Agenda
  - Introduction
  - The Node class
  - The UnorderedList ADT
  - Comparing Implementations
- Reference:
  - Textbook:
    - Problem Solving with Algorithms and Data Structures
      - □ Chapter 3 – Lists
      - □ Chapter 3 – Unordered List Abstract Data Type
      - □ Chapter 3 – Implementing an Unordered List: Linked Lists

Review

- We have used Python lists to implement the abstract data types presented (Stack and Queue)
  - The list is a powerful, yet simple, collection mechanism that provides the programmer with a wide variety of operations
- A Python list stores each element in contiguous memory if possible
  - This makes it possible to access any element in O(1) time
  - However, insertion or deletion elements at the beginning of the list takes O(n)

17.1 Introduction

ADT List

- A list is a collection of items where each item holds a **relative position** with respect to the others
  - We can consider the list as having a first item, a second item, a third item, and so on
  - We can also refer to the **beginning** of the list (the first item) and the **end** of the list (the last item)
- Unordered Vs Ordered
  - Unordered meaning that the items are not stored in a sorted fashion

```
54, 26, 93, 17, 77 and 31
```

```
17, 26, 31, 54, 77 and 93
```

- A Python list ([]) is an implementation of an unordered list, 17.1 Introduction

ADT List

- A list is a collection of items where each item holds a **relative position** with respect to the others
  - We can consider the list as having a first item, a second item, a third item, and so on
  - We can also refer to the **beginning** of the list (the first item) and the **end** of the list (the last item)
- Unordered Vs Ordered
  - Unordered meaning that the items are not stored in a sorted fashion
- A Python list ([]) is an implementation of an unordered list,

17.1 Introduction

ADT List

- What are the operations which can be used with a List Abstract Data?
  - List()
    - Creates a new list that is empty
    - It needs no parameters and returns an empty list.
  - add(item)
    - Adds a new item to the list
    - It needs the item and returns nothing
    - **Assume** the item **is not already** in the list
  - remove(item)
    - Removes the item from the list
    - It needs the item and modifies the list
    - **Assume** the item is **present** in the list

No checking is done in the implementation

17.1 Introduction

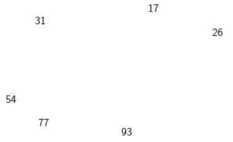
ADT List

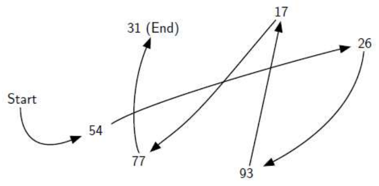
- What are the operations which can be used with a List Abstract Data?
  - search(item)
    - Searches for the item in the list
    - It needs the item and **returns** a boolean value
  - is_empty()
    - Tests to see whether the list is empty
    - It needs no parameters and **returns** a boolean value
  - size()
    - Returns the number of items in the list
    - It needs no parameters and **returns** an integer

17.1 Introduction

Contiguous Memory

- A Python list stores each element in contiguous memory if possible
- List ADT – there is no requirement that the items be stored in contiguous memory
- In order to implement an unordered list, we will construct what is commonly known as a linked list
  - A Node object will store the data in the node of the list
  - A Link to the next Node object

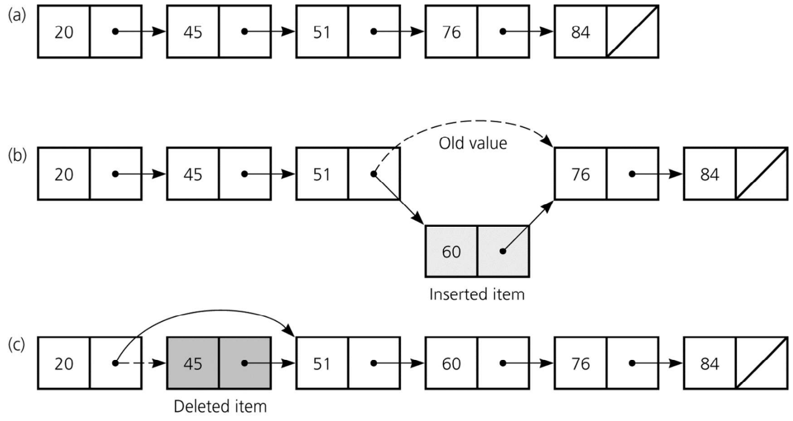




17.1 Introduction

Insertion and Deletion

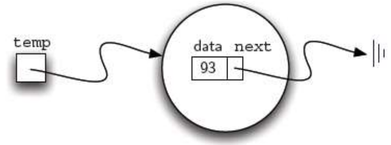
- Items can be inserted into and deleted from the linked list without shifting data

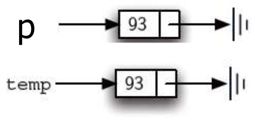


17.2 The Node Class

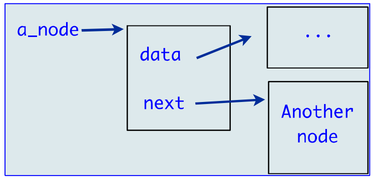
The Node class

- A node is the basic building block of a linked list
  - It contains the data as well as a link to the next node in the list





```
p = Node(93)
temp = Node(93)
```

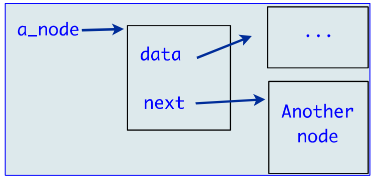


17.2 The Node Class

The Node class

- Code

```
class Node:
    def __init__(self, init_data):
        self.data = init_data
        self.next = None
    def get_data(self):
        return self.data
    def get_next(self):
        return self.next
    def set_data(self, new_data):
        self.data = new_data
    def set_next(self, new_next):
        self.next = new_next
```

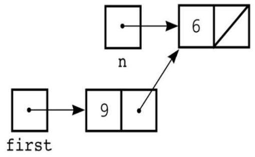


17.2 The Node Class

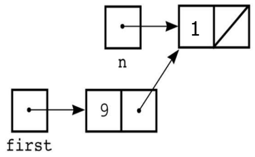
Chain of nodes

- Code

```
n = Node(6)
first = Node(9)
first.set_next(n)
```



```
n.set_data(1)
print(first.get_next().get_data())
```



'1' is displayed

Exercise 1

- What is the output of the following program?

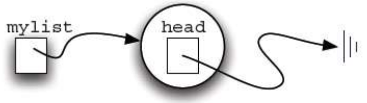
```
def print_chain(n):
    while not n == None:
        print(n.get_data(), end = " ")
        n = n.get_next()

n5 = Node(15)
n6 = Node(34)
n7 = Node(12)
n8 = Node(84)
n6.set_next(n5)
n7.set_next(n8)
n8.set_next(n6)
n5.set_next(None)
```

```
print_chain(n5)
print()
print_chain(n6)
print()
print_chain(n7)
print()
print_chain(n8)
print()
```

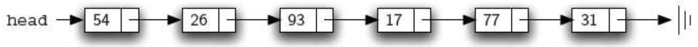
17.3 The UnorderedList Class

The UnorderedList ADT

- The **unordered list** is built from a collection of nodes, each linked to the next by explicit references
  - It must maintain a reference to the first node (head)
  - It is commonly known as a **linked list**
- Examples:
  - An Empty List:



  - A linked list of integers:



17.3 The UnorderedList Class

Operations

- List()
  - Creates a new list that is empty
  - It needs no parameters and returns an empty list
- add(item)
  - Adds a new item to the list
  - It needs the item and returns nothing
  - **Assume** the item **is not already** in the list
- remove(item)
  - Removes the item from the list
  - It needs the item and modifies the list
  - **Assume** the item is **present** in the list

No checking is done in the implementation

17.3 The UnorderedList Class

Operations

- search(item)
  - Searches for the item in the list
  - It needs the item and **returns** a boolean value
- is_empty()
  - Tests to see whether the list is empty
  - It needs no parameters and **returns** a boolean value
- size()
  - Returns the number of items in the list
  - It needs no parameters and **returns** an integer

17.3 The UnorderedList Class

Constructor

- The constructor contains
  - A head reference variable
    - References the list’s first node
    - Always exists even when the list is empty

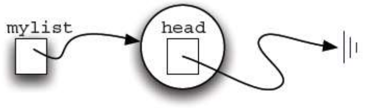
```
class UnorderedList:
    def __init__(self):
        self.head = None
    ...
```

17.3 The UnorderedList Class

Constructor

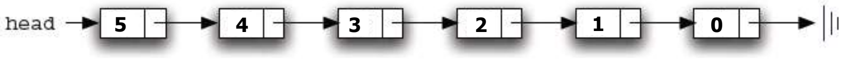
- Example:
  - An Empty List:

```
my_list = UnorderedList()
```



  - A linked list of integers

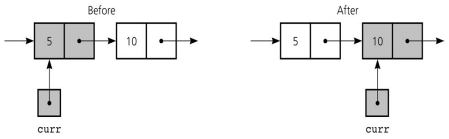
```
my_list = UnorderedList()
for i in range(6):
  my_list.add(i)
```



17.3 The UnorderedList Class

List Traversals

- To traverse a linked list, set a pointer to be the same address as head, process the data in the node, move the pointer to the next node, and so on



17.3 The UnorderedList Class

List Traversals

- Loop stops when the next pointer is None
  - Use a reference variable: curr
    - References the current node
    - Initially references the first node (head)

```
curr = self.head
```

    - To advance the current position to the next node

```
curr = curr.get_next()
```

    - Loop:

```
curr = self.head
while curr != None:
    ...
    curr = curr.get_next()
```

17.3 The UnorderedList Class

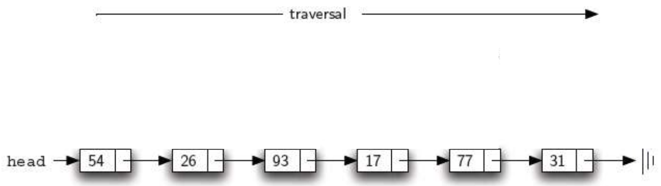
Displaying the Contents

- Traversing the Linked List from the Head to the End
  - Use a reference variable: curr

```
curr = self.head
while curr != None:
    print(curr.get_data(), end=" ")
    curr = curr.get_next()
```

Print the content of a linked list

```
54 25 93 17 77 31
```



17.3 The UnorderedList Class

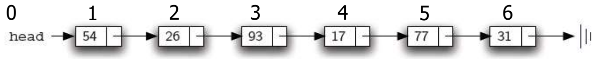
is_empty() & size()

- is_empty()
  - Tests to see whether the list is empty

```
return self.head == None
```

- size()
  - Returns the number of items in the list
  - Traverses the list and counts the number of items

```
curr = self.head
count = 0
while curr != None:
    count = count + 1
    curr = curr.get_next()
```

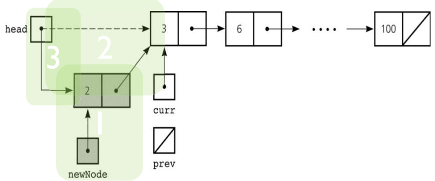


17.3 The UnorderedList Class

Inserting a Node

- To insert at the beginning of a linked list
  - Create a new Node and store the new data into it
  - Connect the new node to the linked list by changing references
    - Change the next reference of the new node to refer to the old first node of the list
    - Modify the head of the list to refer to the new node

```
1 new_node = Node(item)
2 new_node.set_next(self.head)
3 self.head = new_node
```

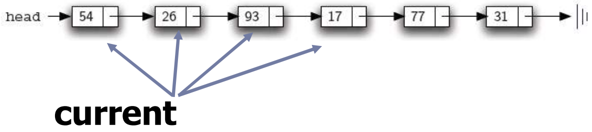


17.3 The UnorderedList Class

Searching an Item

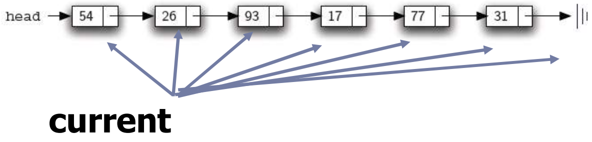
- Searches for the item in the list
  - Returns a Boolean
- Examples:

```
print (my_list.search(17))
```



```
True
```

```
print (my_list.search(1))
```



```
False
```

17.3 The UnorderedList Class
Searching an Item

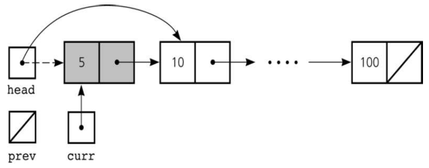
- To search an item in a linked list:
  - Set a pointer to be the same address as **head**
  - Process the data in the node, (search) move the pointer to the **next** node, and so on
  - Loop stops either
    - The item is **found**
    - The next pointer is **None**

```
curr = self.head
while curr != None:
    if curr.get_data() == item:
        return True
    else:
        curr = curr.get_next()
return False
```

17.3 The UnorderedList Class
Deleting a Node

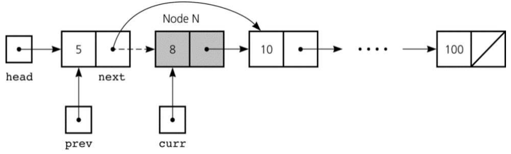
- Removes the item from the list
  - It needs the item and modifies the list
  - Assume the item is present in the list
- Examples:
  - Delete the first node

```
my_list.remove(5)
```



  - Delete a node in the middle of the list
    - With prev and curr references

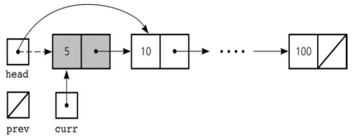
```
my_list.remove(8)
```



17.3 The UnorderedList Class
Deleting a Node

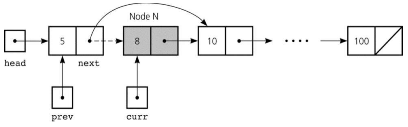
- To delete a node from a linked list
  - Locate the node that you want to delete (**curr**)
  - Disconnect this node from the linked list by changing references
- Two situations:

```
self.head = curr.get_next()
```



  - To delete the first node
    - Modify head to refer to the node after the current node
  - To delete a node in the middle of the list
    - Set **next** of the prev node to refer to the node after the current node

```
previous.set_next(curr.get_next())
```



17.3 The UnorderedList Class
Example

- Example:

```
def TestUnorderedList():
    my_list = UnorderedList()
    number_list = [31, 77, 17, 93, 26, 54]
    for num in number_list:
        my_list.add(num)
        print (my_list.size())
    print (my_list.search(17))
    print (my_list.search(1))
    my_list.remove(31)
    my_list.remove(54)
    print (my_list.size())
```

```
6
True
False
2
```

Exercise 2

- What is the output of the following program?

```
def TestUnorderedList():
    my_list = UnorderedList()
    number_list = [11, 17, 7, 3, 26, 54, 2]
    for num in number_list:
        my_list.add(num)
        print (my_list.size())
    print (my_list.search(17))
    print (my_list.search(1))
    my_list.remove(2)
    my_list.remove(54)
    print (my_list.size())
```

Summary

- Reference variables can be used to implement the data structure known as a linked list
- Each reference in a linked list is a reference to the next node in the list
- Any element in a list can be accessed directly; however, you must traverse a linked list to access a particular node
- Items can be inserted into and deleted from a reference-based linked list without shifting data