Linked Lists

• Linked List vs Array vs Python List
• Singly Linked List
• Bag ADT Revisited
• Building Linked Lists
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.
Step 1: Store the Data

• create a class, Node, to store the data

```python
class Node:
    def __init__(self, data):
        self.data = data
```

• create 3 objects: Node(4), Node(1), Node(7)
• also create 3 variables: a, b, c

```python
a = Node(4)
b = Node(1)
c = Node(7)
```
Step1: Store the Data

class Node:
  def __init__(self, data):
    self.data = data

When I do the command `a = Node(4)`

I create the following object...

`a` is *an alias*, or *another name for* `Node(4)`

`a.data` and `Node(4).data` both refer to ...
Step 2: Add a Second Attribute *Next*

- add a second attribute to the Node class, called *next*
- set its value to `None`

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

```python
a = Node(4)
b = Node(1)
c = Node(7)
```
Step 2: Add a Second Attribute *Next*

- *Rather than* have the variables `a`, `b`, and `c` to refer to the objects `have` the objects point to each other

  ```
a.next = b
b.next = c
```

- *a.next* is now an *alias* or *refers to the same object* that `b` does
Step 2: Add a Second Attribute *Next*

- We can now *get rid of* variables $b$ and $c$, because we no longer need them.
- Refer to the next object in the list using the *next* attribute, i.e. use $a.next$ instead of $b$.
- Now you *no longer have* one variable for each data item in the list.

![Diagram](image)
Node Definition

- The nodes are constructed from a simple storage class:

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

- Many different configurations are possible:
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.
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.
Traversing a Linked List

Given the head reference, we can traverse the nodes.

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

```plaintext```
Searching

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

```python
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

```python
# 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.

```python
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
```
The Bag ADT Revisited

- We can implement the Bag ADT using a linked list.
  - The items will be prepended to the list.
  - Must keep track of the size of the list.
Bag ADT – Linked List

```python
class Bag:
    def __init__(self):
        self._head = None
        self._size = 0

    def __len__(self):
        return self._size

    def __iter__(self):
        return _BagIterator(self._head)

# Defines a private storage class for creating list nodes.
class _BagListNode(object):
    def __init__(self, item):
        self.item = item
        self.next = None
```

Bag ADT – Linked List

class Bag:
# ...

    def __contains__( self, target ):
        curNode = self._head
        while curNode is not None and curNode.item != target:
            curNode = curNode.next
        return curNode is not None

    def add( self, item ):
        newNode = _BagListNode( item )
        newNode.next = self._head
        self._head = newNode
        self._size += 1
class Bag:
  # ...

  def remove( self, item ):
    predNode = None
    curNode = self._head
    while curNode is not None and curNode.item != item :
        predNode = curNode
        curNode = curNode.next

    # The item has to be in the bag to remove it.
    assert curNode is not None, "The item must be in the bag."

    # Unlink the node and return the item.
    self._size -= 1
    if curNode is self._head :
        self._head = curNode.next
    else :
        predNode.next = curNode.next
    return curNode.item
## Comparing Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Python List</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>b = Bag()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>n = len(b)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>x in b</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>b.add(x)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>b.remove(x)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>traversal</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Building Linked Lists

- There are more ways to build a linked list than simply prepending nodes to the front.
  - Appending nodes
  - Sorted linked lists
Using a Tail Reference

- Some applications require items be appended to the end of the linked list.
  - **tail reference** – a second external reference indicating the tail or last node in the list.
Appending Nodes

- Must manage the tail reference as nodes are added/removed.
- **Example**: append 21 to the list.
Appending Nodes

- Given the head and tail reference, we can add an item to a linked list.

```python
newNode = ListNode( item )
if head is None :
    head = newNode
else :
    tail.next = newNode
tail = newNode
```

What is the time complexity to append a node to a linked list, if no tail reference is used?
Removing Nodes

- If the tail node is removed, the tail reference has to be adjusted.
Removing Nodes

- Given the head and tail reference, we can remove a node from a linked list.

```python
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
if curNode is tail :
    tail = predNode
```
The Sorted Linked List

- The items in a linked list can be maintained in sorted order.
Sorted List: Searching

- Searching a sorted list is similar to that of an unsorted list.

```python
def sortedSearch( head, target ):
    curNode = head

    # Stop early when a larger value is encountered.
    while curNode is not None and curNode.data < target :
        if curNode.data == target :
            return True
        else :
            curNode = node.next
    return False
```
Sorted List: Insert

- Adding a new node to a sorted list requires locating the correct position within the list.
  - Locating the position is similar to the removal operation.
  - Use a second temporary reference for the predecessor.

- There are 3 possible cases.
  - front
  - middle
  - back
Sorted List: Insert

- (1) Insert at the front.
Sorted List: Insert

- (2) Insert in the middle.
Sorted List: Insert

- (3) Insert at the end.