Recipe for provider/plan

Choosing among data structures (provider/plan):

1. Create pseudocode of implementations of the ADT using various data structures, using the pseudocode interface.

2. Analyze data structures.
A brief discussion of memory

You can view memory as a sequence of **cells**, each with a location or **address**. Choosing an identifier for a variable associates a name with an address.

A **block** of memory is a contiguous sequence of cells. Different data types (e.g. numbers, strings, objects) might require different block sizes (that is, different numbers of cells).

A cell can contain the address of another cell; this is a **pointer**.

Two main ways of storing multiple pieces of data:

- **Contiguous**: the blocks are all adjacent in memory
- **Linked**: each block has a pointer to the next block (or pointers to next blocks)

A data structure can be viewed as a logical organization of computer memory, typically exploiting patterns of addresses of memory cells.
Arrays

An **array** consists of a block of memory storing multiple items. Each item is stored in a **cell**.

To find one of the items it suffices to have the address/name of the array and the **index** of the cell that contains it, e.g. $T[i]$.

(Note: This is not the same as the Python data type array.)

Key advantage: An array permits immediate access to any item (**random access**).

Key disadvantage: Fixed size can lead to running out of space or wasting space.
A **linked list** consists of not-necessarily-contiguous blocks of memory, where each block contains an item and a pointer to the next item in the list.

Key advantage: It is easy to adjust number of items stored or to rearrange parts of the list.

Key disadvantage: Finding a particular item requires following pointers through all preceding items.
## ADT Multiset pseudocode interface

<table>
<thead>
<tr>
<th>Name</th>
<th>Returns</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateMultiset()</td>
<td>a new empty multiset</td>
<td></td>
</tr>
<tr>
<td>IsEmptyMultiset(M)</td>
<td>true if empty, else false</td>
<td></td>
</tr>
<tr>
<td>LookUpMultiset(M, item)</td>
<td>true if present, else false</td>
<td></td>
</tr>
<tr>
<td>AddToMultiset(M, item)</td>
<td></td>
<td>add item</td>
</tr>
<tr>
<td>DeleteFromMultiset(M, item)</td>
<td></td>
<td>delete item</td>
</tr>
</tbody>
</table>
Linked implementations

head

3 2 6 1 3 5

head

3 4 6 12 20 54

new

5

head

3 2 6 1 3 5

previous

current

head

3 2 6 1 3 5
ADT Multiset code interface

(Available on website as sample 5 in style guide.)

class Multiset:
    ## Multiset() produces a newly
    ## constructed empty multiset.
    ## __init__:  None -> Multiset
    def __init__(self):

        ## self.empty() produces True if self is empty.
        ## empty:  Multiset -> Bool
        def empty(self):
ADT Multiset code interface continued

```python
## value in self produces True if
## value is an item in self.
## __contains__: Multiset Any -> Bool
def __contains__(self, value):

## self.add(value) adds value to self.
## Effects: Mutates self.
## add: Multiset Any -> None
def add(self, value):

## self.delete(value) removes an
## item with value from self,
## if any.
## Effects: Mutates self.
## delete: Multiset Any -> None
def delete(self, value):
```
class Myarray:
    
    # Fields: items is a list of items
    # size is the size of the array
    
    ## Myarray() produces a newly constructed empty array.
    ## __init__: Int -> Myarray
    def __init__(self, size):
        self.items = []
        self.size = size
        for index in range(size):
            self.items.append(None)
More methods for Myarray

```python
## self.access(index) produces the data item with given index.
## access: Myarray, Int -> Any
## Requires: 0 <= index < self.size
def access(self, index):
    return self.items[index]

## self.replace(index, item) replaces the data item with given index.
## Effects: Mutates self.
## replace: Myarray, Int, Any -> None
## Requires: 0 <= index < self.size
def replace(self, index, item):
    self.items[index] = item
```
Making methods safe

```python
## self.replace(index, item) replaces the data
## item with given index or produces
## a warning string.
## Effects: Mutates self if index in range.
## replace: Myarray, Int, Any ->
##      (anyof "Out of range" None)
def replace(self, index, item):
    if index < 0 or index >= self.size:
        return "Out of range"
    else:
        self.items[index] = item
```
class Node:
    
    Fields: item stores any value
    next points to the next node in the list

    ## Node() produces a newly constructed empty node.
    ## __init__: None -> Node
    def __init__(self, item, next = None):
        self.item = item
        self.next = next
Coding Multiset as a linked list

from node import *

class Multiset:
    '''
    Field:  _head points to the first node in the linked list
    '''

    # Multiset() produces a newly constructed empty multiset.
    # __init__:  -> Multiset
    def __init__(self):
        self._head = None
Coding Multiset as a linked list, continued

```python
## value in self produces True if
## value is an item in self.
## __contains__: Multiset Any -> Bool
def __contains__(self, value):
    current = self._head
    while current != None:
        if current.item == value:
            return True
        else:
            current = current.next
    return False
```

See course site for rest.
Coding use of ADT Multiset

Note: Can use any of the files for implementing Multiset (change after “from”).

```python
from multisetll import Multiset

data_set = Multiset()
data_file = open("data.txt","r")
data_list = data_file.readlines()
data_file.close()
for line in data_list:
    data_set.add(line.strip())

value = input("Guess a value or type stop: ")
empty = False
```
while value != "stop" and not empty:
    if value in data_set:
        print("Yes, the set contains", value)
        data_set.delete(value)
    else:
        print("No, the set does not contain", value)
    if data_set.empty():
        print("Sorry, there are no values left to guess.")
        empty = True
    else:
        value = input("Guess a value or type stop: ")
No order of any kind Group A

<table>
<thead>
<tr>
<th>Form of the data</th>
<th>single values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of the data</td>
<td>any</td>
</tr>
<tr>
<td>Information relating data</td>
<td>none</td>
</tr>
<tr>
<td>Positioning of the data</td>
<td>none</td>
</tr>
</tbody>
</table>
## ADT Set pseudocode interface

Note: Add results in a change only if the item is not already in S.

<table>
<thead>
<tr>
<th>Name</th>
<th>Returns</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateSet()</td>
<td>a new empty set</td>
<td></td>
</tr>
<tr>
<td>IsEmptySet(S)</td>
<td>true if empty, else false</td>
<td></td>
</tr>
<tr>
<td>LookUpSet(S, item)</td>
<td>true if present, else false</td>
<td></td>
</tr>
<tr>
<td>AddToSet(S, item)</td>
<td></td>
<td>add item</td>
</tr>
<tr>
<td>DeleteFromSet(S, item)</td>
<td></td>
<td>delete item</td>
</tr>
</tbody>
</table>
Definitions

In successful search the item is present; in unsuccessful search the item is not.

An algorithm is comparison-based if the only types of actions performed on data items are comparisons.

Data structures may be internal (stored entirely in memory) or external (making use of external storage).

A sorting algorithm is stable if equal elements end up in the same relative order as in the input.