# CS 234: Data Types and Structures

Naomi Nishimura

Module 3

Date of this version: September 26, 2019

**WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.**

**WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.**

# Recipe for provider/plan

Recipe for provider/plan (choosing among implementations):

1. Create pseudocode of various options for data structures (or ADTs) and algorithms to implement the ADT and its operations.
2. Analyze the costs of each operation for each implementation.
3. Provide options for packages of operation costs.

*Note: The addition of "(or ADTs)" should be part of the first step in the recipe, both in Module 1 and in the mini-textbook. Since the final versions of those have already been released, the note about the change is being made here instead.*

# A brief discussion of memory

A **cell** stores a single bit (0 or 1).

A **chunk** of memory is a contiguous sequence of cells. (See the mini-textbook for a discussion of this non-standard term.)

Different data types (e.g. numbers, strings, objects) might require different chunk sizes.

The location of a cell (or the first cell in a chunk) is its **address**. A **pointer** is a type of data that represents an address.

When a program requests memory for a variable, the operating system finds a chunk of free memory (not currently allocated to any program) and associates the name of the variable with the address of the chunk.

# Storing multiple pieces of data

For a group of data items, typically all of the same type, there are two main options:

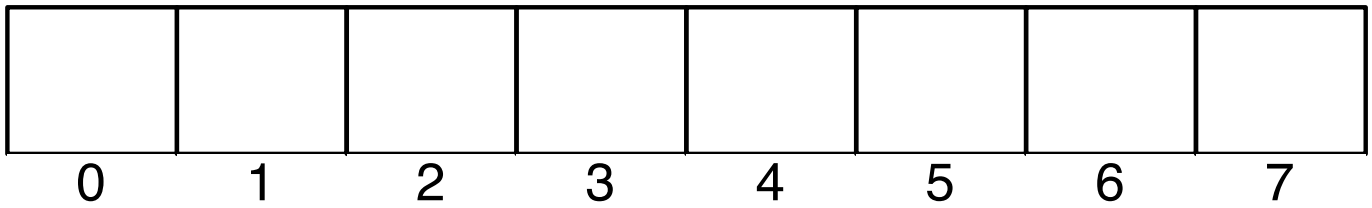
- **Contiguous**: Use one chunk for the entire group. Memory for each individual data item in the group can be viewed as a **subchunk**.
- **Linked**: Use one chunk for each data item. The chunk storing one data item may contain one or more pointers to other chunks.

When the data items are **compound data**, consisting of multiple **fields**, a chunk or subchunk might be further divided into subchunks for each field.

# Data structure: Array (contiguous)

An **array** is a chunk of memory that can be divided into a sequence of **slots**, each of which can store a data item or **element**.

The **size** of the array is the number of slots. The total number of bits in the chunk is the product of the size of the array and the number of bits per slot (or **subchunk**).




To access a slot, it suffices to have the address/name of the array and the **index** (that is, its location in the sequence of elements), e.g. `T[i]`.

Costs: $\Theta(1)$ to allocate memory for an array; $\Theta(1)$ to read or write a single data item.

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

# ADT Multiset

The ADT Multiset can store multiple copies of data items.

Preconditions: For all *M* is a multiset and *Data* is any data item; for *DELETE Data* must be in *M*.

Postconditions: Mutation by *ADD* (adds one copy of *Data*) and *DELETE* (deletes one copy of *Data*).

| Name | Returns |
|---|---|
| *CREATE()* | a new empty multiset |
| *IS_IN(M, Data)* | *True* if present, else *False* |
| *ADD(M, Data)* | |
| *DELETE(M, Data)* | |

# Array implementations of Multiset

**Implementing *CREATE***

Options:

- Create an array of size large enough to store as many data items as may ever be stored or create a smaller array, eventually replacing it with a bigger one as needed.
- Initialize all slots in the array to be empty or keep track of which ones store data items so that only those are read.

Not options:

- Creating a small array and running out of space
- Reading slots that have not been initialized or written

Key idea: Initialization or other writing is needed before reading.

# Subtasks for *Is_In*, *Add*, and *Delete*

For *Is_In*, *Add*, and *Delete*, we need to do the following:

- **search** for the item (for *Is_In* and *Delete*) or a slot to add an item (for *Add*)
- **modify** the array (for *Add* and *Delete*)

Searching in an array:

- For *Add*, we can place the new data item in the first empty slot found.
- For *Is_In* and *Delete*, we can use a loop to try each slot in turn.

Note: For now, we make no assumptions about our data items, so we can not assume that we can store them in order by value.

# Modifications to data in an array

Easy modifications:

- Add a new element to an empty slot
- Delete an element, resulting in an empty slot

Not-so-easy modifications:

- Add a new element to an array that is already full
- Ensure that all nonempty slots come before all empty slots

In the implementations we consider, let $k$ be the number of items stored when the operation is executed and let $s$ be the size of the array.

# Contiguous implementation 1

Data structure: Array of size $s$

Algorithms:

- *CREATE* initializes all slots to empty
- *IS_IN* searches slots in order by increasing index
- *ADD* searches for the first empty slot
- *DELETE* searches, then rewrites slot to empty

Key idea: All slots can be searched by looping through indices.

# Contiguous implementation 2

Idea: Keep track of empty space to make *ADD* faster.

Data structures:

- Array of size *s*
- Variable *First* storing index of first empty slot or *"Full"*

Algorithms:

- *CREATE* initializes all slots to empty, sets *First* to 0
- *IS_IN* searches slots in order by increasing index
- *ADD* adds new data item to *First*, updates *First*
- *DELETE* searches, then rewrites slot to empty; updates *First* if needed

Key idea: Implementations of all operations need to ensure that all data structures retain intended properties.

# Contiguous implementation 3

Idea: Start with a small array, replacing it with a bigger array as needed.

Data structure: Array of size dependent on number of items stored

Algorithms:

- *ADD* creates a new larger array if number of items reaches some upper threshold on fraction of slots, copies over items
- *DELETE* creates a new smaller array if number of items reaches some lower threshold on fraction of slots, copies over items

Key idea: Contiguous memory has a fixed size.

Observations:

- Saves space
- May provide improvement over contiguous implementation 2 for *IS_IN* and *DELETE* as worst-case cost will depend on size of the current array
- Idea used in coding of Python lists

# Contiguous implementation 4

Idea: Arrange so all nonempty slots followed by all empty slots.

Data structures:

- Array of size $s$ with all data items stored contiguously, starting at index 0
- Variable *First* storing index of first empty slot

Algorithms:

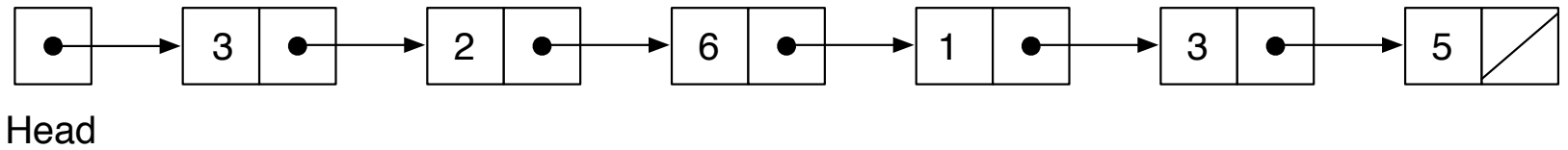
- *CREATE* does not initialize the array; *First* set to 0
- *IS_IN* searches slots before *First* in order by increasing index
- *ADD* adds new data item to *First*, updates *First*
- *DELETE* searches, rewrites with last data item, updates *First* to one smaller

Key idea: Use a variable to store an index indicating the boundary between empty and full slots.

# Data structure: Linked list

A **linked list** consists of not-necessarily-contiguous **linked nodes** (simplified at times to the term **nodes**), where each node is a chunk of memory containing data and a pointer to the next node in the linked list.

In our diagrams, pointers will be represented by arrows, and empty or **null** pointers by diagonal lines.




We will be careful to distinguish between a pointer and the data to which a pointer points.

Costs: $\Theta(1)$ to allocate memory for a node; $\Theta(1)$ to read or write a single data item or a single pointer.

# Linked implementation

Idea: Use a linked list.

Data structure:

- Variable *Head* storing a pointer to a linked list of nodes, each storing a data item and a pointer to the next node in the list

Algorithms:

- *CREATE* sets variable *Head* to null pointer
- *IS_IN* searches nodes in order
- *ADD* adds to beginning of the list
- *DELETE* searches, then removes node

Key idea: Size of linked memory is flexible.

# Subtasks for *Is_In*, *Add*, and *Delete*

For *Is_In*, *Add*, and *Delete*, we need to do the following:

- **search** for the item (for *Is_In* and *Delete*)
- **modify** the linked list (for *Add* and *Delete*)

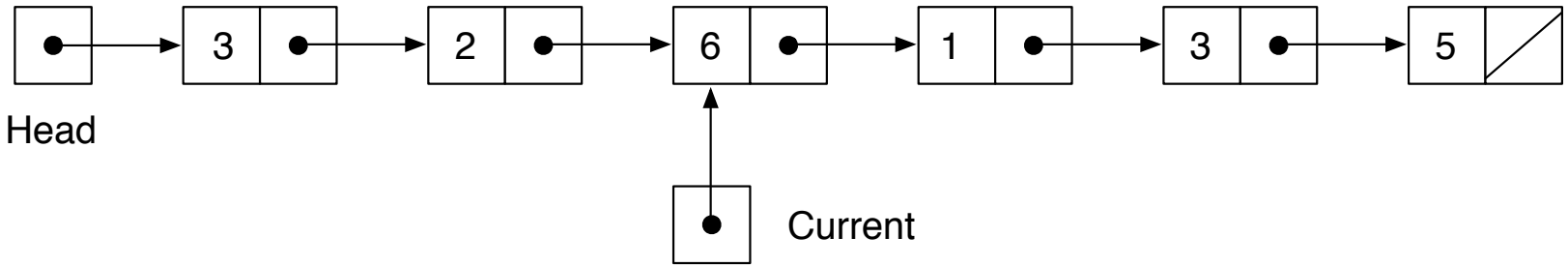
Searching in a linked list

- For *Add*, we can add the new item at the beginning of the list.
- For *Is_In* and *Delete*, we can use a loop to try each node in turn.

# Search

Loop through the nodes:

- Set the pointer in *Current* to equal the pointer in *Head*.
- If *Current* is the null pointer, stop.
- Compare the data in the node to which *Current* points to the value being searched.
- If the data and search value are not equal, set *Current* to the pointer in the node to which *Current* points and repeat.



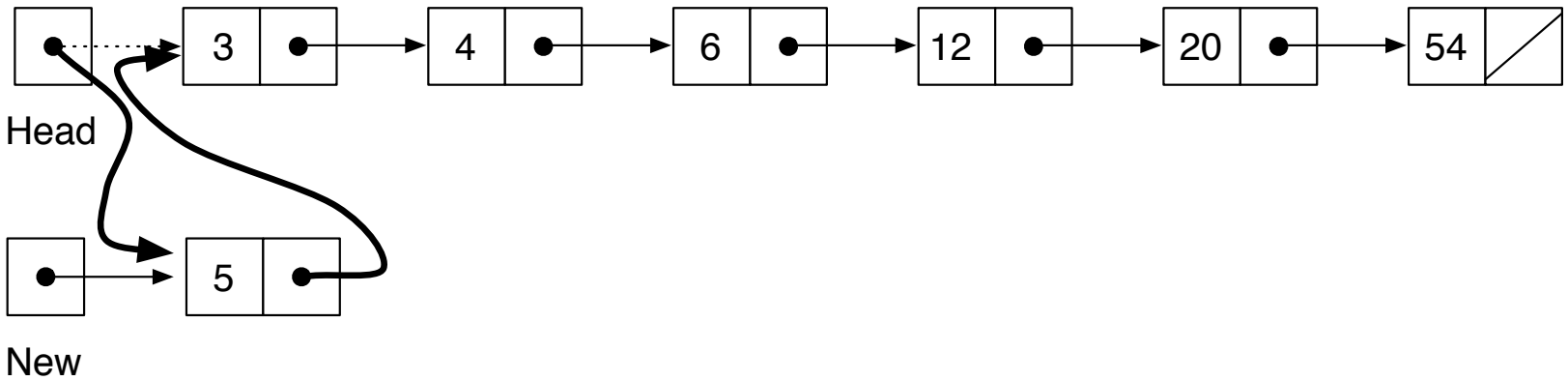

Key idea: Use a pointer to the current linked node.

Key idea: Search in linked memory using pointers.

# Modification to add a node

Splice in a node at the beginning of a list:

- Create a new pointer *New* that points to a new node containing *Data*.
- Set the pointer in the node to which *New* points to equal the pointer in *Head*.
- Set the pointer in *Head* to equal the pointer in *New*.

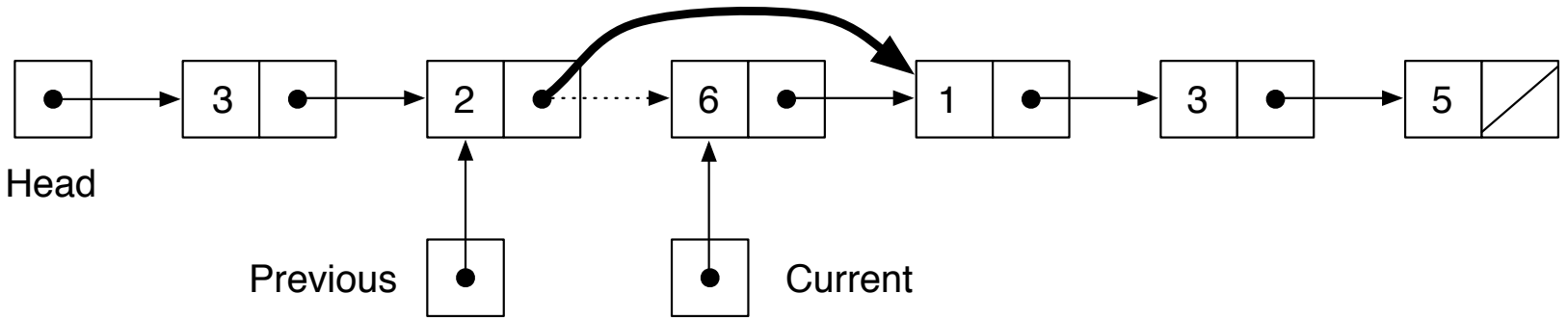



Key idea: Changing the number of nodes in a linked implementation is easy.

# Modification to remove a node

Splice out a node:

- Using search, move *Current* to the node to delete and *Previous* to the previous node.
- Set the pointer in the node to which *Previous* points to equal the pointer in the node to which *Current* points.
- In the special case in which *Current* is equal to *Head*, instead set *Head* to equal the pointer in the node to which *Current* points.




Key idea: Use two pointers for deletion.

# Summary of options

**Provider/plan step 3**: Provide options for packages of operation costs

Worst-case running time as a function of $k$ (number of items stored)

| | **Array (4)** | **Linked list** |
|---|---|---|
| *CREATE* | $\Theta(1)$ | $\Theta(1)$ |
| *IS_IN* | $\Theta(k)$ | $\Theta(k)$ |
| *ADD* | $\Theta(1)$ | $\Theta(1)$ |
| *DELETE* | $\Theta(k)$ | $\Theta(k)$ |

Notes:

- User does not need to know name of data structure to make a choice.
- Summary charts will not be provided for future ADTs, but you should consider making your own.
- Not all ADTs will have data structures with such similar behaviour.

# Code interfaces in Python

User/code:

- Agree on the code interface.
- Code a solution to the problem using the ADT.

Provider/code:

- Agree on the code interface.
- Code the chosen data structure and algorithms implementing the ADT.

Classes review:

- `__init__` - this can be used for creating an ADT
- `self` - used in methods to refer to item itself
- `__contains__` - used for `in`, operator overloading
- `__eq__` - used for $==$ and by default for $!=$
- `__str__` - string represention used by `print`
- `__repr__` - string representation used for tests

# Design recipe

Previous courses:

- Some steps used in planning
- Some steps evident in comments
- Some steps evident in code

CS 234:

- Choice of ADTs is a new planning step.
- Preconditions and postconditions of implementations of operations are handled by contract and purpose of functions.
- See style guide for review of best practices.
- Marks will focus on user/provider division and preconditions and postconditions.
- Use of examples and tests is encouraged for good form and partial marks, but will not be required.

# ADT Multiset code interface

**User/code and provider/code step 1**: Agree on the code interface.
(Available on website as sample 5 in style guide.)

```
class Multiset:
    ## Multiset() produces a newly
    ##       constructed empty multiset.
    ## __init__:  -> Multiset
    def __init__(self):
```

# ADT Multiset code interface: *Is_In* and *Add*

```
## 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 by adding value to self.
## add:  Multiset Any -> None
def add(self, value):
```

# ADT Multiset code interface: *DELETE*

```
## self.delete(value) removes an
##       item with value from self.
## Effects:  Mutates self by removing an
##       item with value from self.
## delete:  Multiset Any -> None
## Requires:  self contains an item with value value
def delete(self, value):
```

# Coding arrays in Python

Many programming languages have arrays built in. Python doesn't.

Possible approaches:

- Use a Python list instead. (But then other list operations can be used.)
- Use the `ctypes` module to access an array. (But it uses stuff that you are not expected to understand.)
- Use a class, with a Python list limited to array operations. (Not ideal, but best pedagogically.)

Note: As a consequence of our choices, our implementation does not match any of the ones we discussed and analyzed in the planning stage.

# Details of coding arrays in Python

```
class Contiguous:
    """
    Fields: _items is a list of items
            _size is number of items that can be stored
    """
    ## Contiguous(s) produces contiguous memory of size s
    ##  and initializes all entries to None.
    ## __init__:  Int -> Contiguous
    ## Requires:  s is positive
    def __init__(self, s):
        self._items = []
        self._size = s
        for index in range(self._size):
            self._items.append(None)
```

# Methods for Contiguous: `size` and `access`

```
## self.size() produces the size of self.
## size:  Contiguous -> Int
def size(self):
    return self._size

## self.access(index) produces the value at
##      the given index.
## access:  Contiguous Int -> Any
## Requires:  0 <= index < self._size
def access(self, index):
    return self._items[index]
```

# Method for Contiguous: `store`

```
## self.store(index, value) stores value
##       at the given index.
## Effects:  Mutates self by storing value
##       at the given index.
## store:  Contiguous Int Any -> None
## Requires:  0 <= index < self._size
def store(self, index, value):
    self._items[index] = value
```

# Making methods safe

```
## self.safe_store(index, value) stores value
##       at the given index or produces
##       a warning string.
## Effects:  Mutates self by storing value if index in range.
## safe_store:  Contiguous Int Any ->
##           (anyof "Out of range" None)
def safe_store(self, index, value):
    if index < 0 or index >= self._size:
        return "Out of range"
    else:
        self._items[index] = value
```

# Using contiguous implementations in CS 234

The module `contiguous.py` is provided to allow you to simulate the use of an array.

To preserve boundaries in the simulation, you should access a `Contiguous` object only through the code interface of provided methods (discussed in the mini-textbook).

# Coding linked lists in Python

```
class Single:
    """
    Fields: _value stores any value
            _next stores the next node or None, if none
    """
    ## Single(value) produces a newly constructed
    ##       singly-linked node storing value with a
    ##       link to next.
    ## __init__:  Any (anyof Single None) -> Single
    def __init__(self, value, next = None):
        self._value = value
        self._next = next
```

# More methods for linked implementations

The methods allow reading and writing of data items and links in nodes:

- `repr(node)` - produces a string with the value in `node`
- `node.access()` - produces the value stored in `node`
- `node.next()` - produces the node to which `node` is linked, if any, else `None`
- `node.store(value)` - stores `value` in `node`
- `node.link(other)` - links `node` to `other`, which can be either a `node` or `None`

There is also an object `Double` for doubly-linked nodes.

Please see the mini-textbook for details.

# Using linked implementations in CS 234

The module `linked.py` provides objects for both singly-linked and doubly-linked nodes.

To preserve boundaries in the simulation, you should access a `Single` or `Double` object only through the code interface of provided methods (discussed in the mini-textbook).

In your implementations, to create a variable that can store a pointer to a node, you will create a variable that stores a `Single` or a `Double`.

# Coding Multiset as an array

```
from contiguous import *
INFINITY = 100

class Multiset:
    """
    Fields: _data is a one-dimensional array of the items
            _first is the index of the next
                place to fill
    """

    ## Multiset() produces a newly
    ##       constructed empty multiset.
    ## __init__:  -> Multiset
    def __init__(self):
        self._data = Contiguous(INFINITY)
        self._first = 0
```

# More methods for array implementation

```
## value in self produces True if
##      value is an item in self.
## __contains__:  Multiset Any -> Bool
def __contains__(self, value):
    for index in range(self._data.size()):
        if self._data.access(index) == value:
            return True
    return False
```

Please see the website for the complete file.

# Coding Multiset as a linked list

```
from linked import *
class Multiset:
    """
    Fields: _first points to the first node (if any)
                in a singly-linked list
    """
    ## Multiset() produces a newly constructed
    ##       empty multiset.
    ## __init__:  -> Multiset
    def __init__(self):
        self._first = None
```

# Adding a new node using `Single()`

```
## self.add(value) adds value to self.
## Effects:  Mutates self by adding value to self.
## add:  Multiset Any -> None
def add(self, value):
    new_node = Single(value, self._first)
    self._first = new_node
```

# More methods for the linked implementation

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

Please see the website for the complete file.

# Coding use of ADT Multiset

## **User/code step 2**: Code a solution to the problem using the ADT.

Note: Can use either of the files for implementing Multiset (change after "from").

```
from multisetcontiguous import Multiset


## fix_nuggets(birds) replaces each instance
##       of "Chicken nugget" by "Chicken"
## Effects:  Mutates birds by replacing each
##       instance of "Chicken nugget" by "Chicken"
## fix_nuggets:  Multiset -> Multiset
def fix_nuggets(birds):
    while "Chicken nugget" in birds:
        birds.delete("Chicken nugget")
        birds.add("Chicken")
    return birds
```

# Types of data structures

Simple structures include:

- arrays
- arrays used with one or more auxilliary variables
- linked lists
- linked lists used with one or more auxilliary variables
- linked implementations with pointers in both directions

Complex structures include:

- linked structures where nodes store multiple values and/or multiple pointers
- combinations of arrays and linked lists
- high-level organizations of data that in turn can be implemented using ADTs

# Relating ADTs and data structures

Keep in mind:

- Different data structures can implement the same ADT.
- Different ADTs can be implemented using the same data structure.
- Different algorithms can be used to implement the same ADT operation on the same data structure.

Exercise your understanding by trying to mix and match ADTs, data structures, and algorithms.

# Data structure design

Design ideas:

- Use contiguous memory, linked memory, or a mixture
- Use extra variables to store extra information (such as number of data items stored)

Design principles:

- Algorithms must always preserve form and meaning of data structure
- Extra information may make some operations faster but may be costly to maintain

# Replacing a value, revisited

Options to consider:

- Write an algorithm using existing ADT operations.
- Augment the ADT by adding a new operation.

In general:

- The user creates the algorithm without accessing the data structure directly.
- The provider creates an implementation that supports all the operations in the augmented ADT.

For replacing a value:

- We considered the first option in Module 2.
- As an exercise, consider how to write a replace operation for each of the implementations of the ADT Multiset.

# Programming in CS 234

When completing programming questions for the course, please keep the following in mind:

- The purpose of the questions is to allow you to exercise and demonstrate your understanding of course concepts.
- You may not receive full marks for a solution that does not adhere to the spirit of the question or the role that is being played (user or provider).

# Coding in different roles

- When playing the role of the user of an ADT, you should access the ADT only through the methods in the code interface. You will lose marks for looking "under the hood" and directly accessing any data structure (or other ADT) that is used to implement the ADT.
- When playing the role of the provider of the ADT, you should be dealing directly with the data structure (or other ADT) that is used in the implementation. If there is a method that can be implemented using other methods (like a user would do), you will lose marks for using the other methods instead of writing a new one that directly accesses the data structure (or other ADT).

# ADT Set

The ADT Set stores at most one copy of each data item.

Preconditions: For all *S* is a set and *Data* any data item; for *ADD Data* is not in *S*; for *DELETE Data* must be in *S*.

Postconditions: Mutation by *ADD* (adds item) and *DELETE* (deletes item).

| Name | Returns |
|---|---|
| *CREATE()* | a new empty set |
| *IS_IN(S, Data)* | *True* if present, else *False* |
| *ADD(S, Data)* | |
| *DELETE(S, Data)* | |

# Array and linked implementations of ADT Set

Consider modifications on implementations for ADT Multiset

Key difference: A search is required before *ADD*, using *IS_IN* to ensure that the precondition is satisfied.

Notice that a search is needed for each of *IS_IN*, *ADD*, and *DELETE*.

We will return later to the importance of searching.

# Comparison of contiguous and linked memory

Contiguous memory:

- Key advantage: An array permits immediate access to any element by **random access**.
- Key disadvantage: Fixed size can lead to running out of space or wasting space.

Linked memory:

- Key advantage: It is easy to adjust the number of nodes or to rearrange parts of the list.
- Key disadvantage: Finding a particular node requires following pointers through all preceding nodes.

Note: Some of the advantages and disadvantages will become clearer when implementing other ADTs.

# Optimizing implementations for special types of data

Recall that more general ADTs tend to have higher costs for each operation.

Similarly, implementations that can store **general data** (on which the only operations are tests for equality and inequality) may have higher costs than implementations taking advantage of known properties of the data.

# Types of data

Although all data is stored as sequences of 0's and 1's, a particular programming language may only allow certain types of operations on particular kinds of data.

**General data** can only be compared for equality.

**Orderable data** can be compared for equality or ordering ($<$, $>$, $\leq$, and $\geq$), e.g. numbers or strings.

**Digital data** supports computations other than comparisons for equality or ordering, such as:

- Decomposing a string into characters
- Applying arithmetic operations to a number

# Special case: Data items from a fixed range

Suppose all the inputs are integers in the range $0, 1, \ldots, r-1$.

Data structure: Array of size $r$, where each slot is a single bit.

Algorithms:

- *CREATE* initializes all slots to 0.
- *IS_IN* searches in slot *Data*, returning *True* if the value is 1.
- *ADD* sets the slot *Data* to 1.
- *DELETE* sets the slot *Data* to 0.

Analysis:

- *CREATE* $\Theta(r)$
- *IS_IN* $\Theta(1)$
- *ADD* $\Theta(1)$
- *DELETE* $\Theta(1)$

This implementation is called a **bit vector**.

# Module summary

Topics covered:

- Memory
- Data structure: Array
- Array implementations of Multiset
- Data structure: Linked list
- Linked implementation
- Code interfaces in Python
- Coding arrays in Python
- Making methods safe
- Coding linked lists in Python
- Coding use of Multiset
- Types of data structures
- Data structure design
- ADT Set
- Types of data
- Bit vector