COMP1002
Data Structures and Algorithms

Lecture 10: DSA in Practice

Curtin University
Department of Computing

Last updated: October 25, 2020

Copyright Warning

This Week

- Algorithm Selection
- Java Collections
- ADT selection

Algorithm Selection

- Computer Science has been developing and discovering algorithms for over 50 years
- Most problems have been attempted before
- First step should be to survey existing options

- e.g. for your assignment...
  - Graph algorithms: add vertices and edges, get adjacent...
  - Shortest paths
  - Nearby search
  - Sorting a linked list (possibly)

Algorithm Selection

- **What should we consider?**
- Type and size of data and sort keys
- Order – random, semi-sorted
- How often you will search / sort
- Time efficiency
- Space efficiency
- Understandability of algorithm
- How perfect the results have to be?
    - Is a good path as useful as the shortest path?

Algorithms in DSA

- Searching: Linear, Binary
- Sorting: Bubble, Selection, Insertion, Quicksort, Mergesort
- Binary Search Trees: Insert, Delete
- Graphs: Breadth/Depth First Search, Shortest Path
- Hash Tables: Linear/Quadratic/Double, Open Addr
- Heaps: Trickle-up/down, Heapify, Heapsort
- Advanced Trees: Insert, Delete, Split/Merge, "Balance"

More Sorting

– We looked at some excellent visualisations
– Clearly there are even better performing sorts
– How do we go beyond $O(N^{2)}$ and $O(N \log N)$?

– We have looked at "comparison-based" sorts
– There is a cost to every comparison

Visualisations

- Colour
  - https://www.youtube.com/watch?v=h-QYzgTmgVI
- Disparity
  - https://www.youtube.com/watch?v=IjIViETya5k

Shell Sort

- Developed by Donald Shell in 1959
- A variation of Insertion Sort
- Improves handling of values that are far from their final location
- Views the list as an interleaved bundle of lists
- Unstable
- Takes advantage of partially sorted data

Algorithm

– For each gap size (largest downto 1)
  - Consider each list is made up of the elements with a gap "h" between them
  - Sort each interleaved list using insertion sort
  - When all lists for gap "h" are sorted, the overall data is "h-sorted"

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
Input data	62	83	18	53	07	17	95	86	47	69	25	28
After 5-sorting	17	28	18	47	07	25	83	86	53	69	62	95
After 3-sorting	17	07	18	47	28	25	69	62	53	83	86	95
After 1-sorting	07	17	18	25	28	47	53	62	69	83	86	95

- The first pass, 5-sorting, performs insertion sort on five separate subarrays (a1, a6, a11), (a2, a7, a12), etc
- The next pass, 3-sorting, performs insertion sort on the three subarrays (a1, a4, a7, a10), (a2, a5, a8, a11), etc
- The last pass, 1-sorting, is an ordinary insertion sort of the entire array (a1,..., a12).

https://en.wikipedia.org/wiki/Shellsort

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
Input data	62	83	18	53	07	17	95	86	47	69	25	28
After 5-sorting	17	28	18	47	07	25	83	86	53	69	62	95
After 3-sorting	17	07	18	47	28	25	69	62	53	83	86	95
After 1-sorting	07	17	18	25	28	47	53	62	69	83	86	95

- The first pass, 5-sorting, performs insertion sort on five separate subarrays (a1, a6, a11), (a2, a7, a12), etc
- The next pass, 3-sorting, performs insertion sort on the three subarrays (a1, a4, a7, a10), (a2, a5, a8, a11), etc
- The last pass, 1-sorting, is an ordinary insertion sort of the entire array (a1,..., a12).

https://en.wikipedia.org/wiki/Shellsort

Counting Sort

– Sorts keys in a specific range (small ranges preferred)

– Counts the number of occurrences of each key

– Then knows how much space those keys will take and slots them in

– Stable sort, but not in-place

– Takes advantage of duplicates

– O(n+k) – k=range... not a comparison sort

– Often used as a subroutine for other sorts (Radix)

http://www.geeksforgeeks.org/counting-sort/

Counting Sort - Algorithm

- Given array input[]
- Create Count[] and Result[]
- Fill Count[] with the count of each key in input[]
- Update Count[] to store the sum of the previous counts
  - This will give us the position for each group of keys
- Work through input[] backwards, slotting values into Result[], decrementing the count of each, each time

Counting Sort - Example

int input[] = { 2, 1, 4, 5, 7, 1, 7, 11, 8, 9, 10 };

Index	0	1	2	3	4	5	6	7	8	9	10	11
Count[]	0	2	1	0	1	1	0	2	1	1	1	1

Index	0	1	2	3	4	5	6	7	8	9	10	11
Modified Count[]	0	2	3	3	4	5	5	7	8	9	10	11

Count[i]=Count[i] + Count[i-1]

Result[]	0	1	1	2	4	5	7	7	8	9	10	11

Count[input[i]] will tell you the index position of input[i] in Result[]

http://algorithms.tutorialhorizon.com/counting-sort/

Radix Sort

- Counting Sort was limited to a small range of values
- If we have larger range of values, need to adjust...
- Radix Sort uses the Counting Sort as a function
- Works on each digit, sorting from Least Significant Digit (LSD) or Most Significant Digit (MSD)
- Stable sort, uses extra space (through Counting Sort)
- Tricky to calculate complexity – dependent on base (b)
- $O((n+b)*\log_b(k))$ – if $b=n$, $O(n)$

Radix Sort LSD - Example

– Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

– Sorting by least significant digit (1s place) gives:

170, 90, 802, 2, 24, 45, 75, 66

– Sorting by next digit (10s place) gives:

802, 2, 24, 45, 66, 170, 75, 90

– Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

http://www.geeksforgeeks.org/radix-sort/

Visualisations

– Colour
    - https://www.youtube.com/watch?v=h-QYzgTmgVI

– Disparity
    - https://www.youtube.com/watch?v=IjIViETya5k

Choosing Abstract DataTypes

– We now have many ADTs we can work with

Linked Lists, Stacks, Queues,

BSTs, Graphs, Heaps, HashTables,

Red-Black Trees, 2-3-4 Trees, B-Trees

– There are many others

– There are many implementations

Choosing ADTs

- Know your data
- What do you want to use it for?
  - Unique values
  - Order important?
  - Search time important?
  - Static or changing data?
- Prototype – try them out

Java ADTs

– Beyond DSA, you can use the in-built ADTs
– You now know enough about ADTs to understand the collections and make choices
– Which class or interface to use?
– Extend using interfaces, or use Java implementations?
– Build your own?
– Source code is available

Java Collection Documentation

- Always look to the online documentation for up to date information
- These slides are based on the tutorial:
  - https://docs.oracle.com/javase/tutorial/collections/
- Some are taken directly from:
  - http://www.cs.nyu.edu/courses/fall07/V22.0102-002/lectures/JavaCollections.ppt
  - by Prof Evan Korth, NYU

Java Collections

- A collection — sometimes called a container — is simply an object that groups multiple elements into a single unit
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- Typically, they represent data items that form a natural group:
  - a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

Collections Framework

– A collections framework is a unified architecture for representing and manipulating collections.

– All collections frameworks contain the following:

- Interfaces: These are abstract data types that represent collections.

- Implementations: These are the concrete implementations of the collection interfaces. Reusable data structures.

- Algorithms: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. (Polymorphic)

See also: C++ Standard Template Library (STL) and Smalltalk's collection hierarchy

Benefits

– Reduces programming effort:
  - By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.

– Increases program speed and quality:
  - Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.

Benefits

- Allows interoperability among unrelated APIs:
  - The collection interfaces are the vernacular by which APIs pass collections back and forth. Our APIs will interoperate seamlessly, even though they were written independently.
- Reduces effort to learn and to use new APIs:
  - Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections, with standard collection interfaces, the problem went away.

Benefits

– Reduces effort to design new APIs:

  - This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.

– Fosters software reuse:

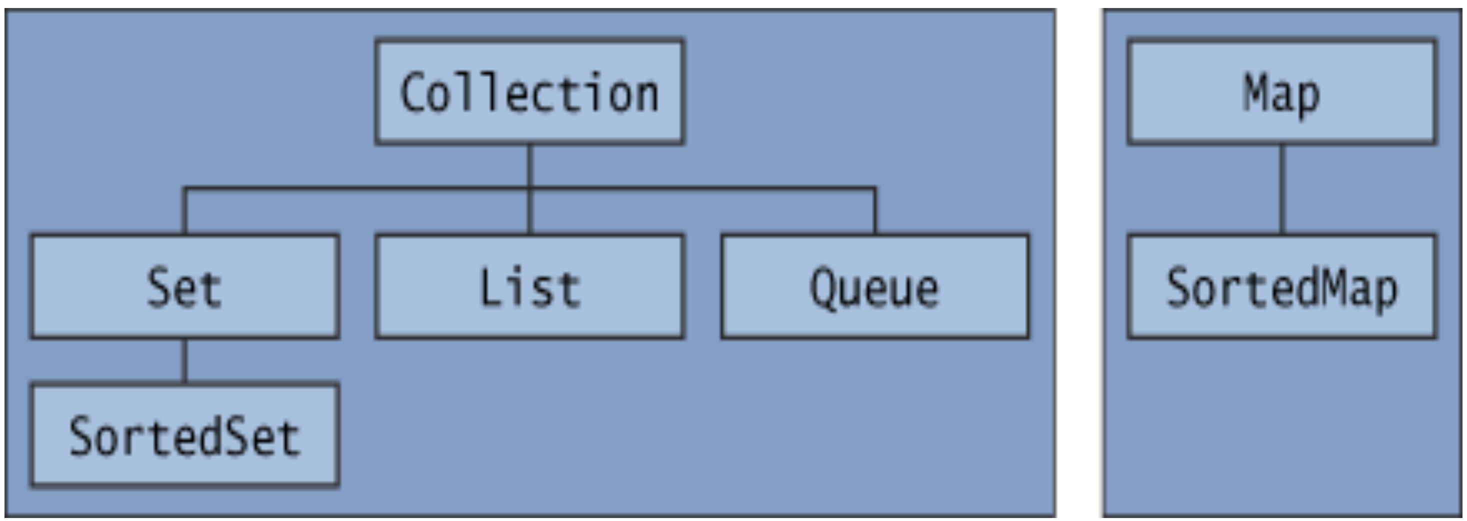
  - New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Collection interfaces

– The core collection interfaces encapsulate different types of collections.

– They represent the abstract data types that are part of the collections framework.

– They are interfaces no implementation provided



public interface Collection<E> extends Iterable<E>

- **Collection** — the root of the collection hierarchy.
- A collection represents a group of objects known as its *elements*.
- The Collection interface is the common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired.
- Some types of collections allow duplicate elements, and others do not.
- Some are ordered and others are unordered.
- The Java platform doesn't provide any direct implementations of this interface

public interface Collection<E> extends Iterable<E>

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

We already know about Iterators...

- An `Iterator` is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.
- You get an `Iterator` for a collection by calling its `iterator()` method.
- The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

public interface **Set<E>** extends Collection<E>

- **Set** — a collection that cannot contain duplicate elements.
- This interface models the mathematical set abstraction and is used to represent sets
- Examples: the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.

public interface **Set<E>** extends Collection<E>

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Note: nothing added to Collection interface – except no duplicates allowed

public interface **List<E>** extends Collection<E>

- **List** — an ordered collection (sometimes called a *sequence*).
- Lists can contain duplicate elements.
- The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
- If you've used Vector, you're familiar with the general flavor of List.

public interface **List<E>**extends Collection<E>

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

A note on ListIterators

- The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces.
- The **hasPrevious** and the **previous** operations are exact analogues of **hasNext** and **next**. The previous operation moves the cursor backward, whereas next moves it forward.
- The **nextIndex** method returns the index of the element that would be returned by a subsequent call to next, and **previousIndex** returns the index of the element that would be returned by a subsequent call to previous
- The **set** method overwrites the last element returned by **next** or **previous** with the specified element.
- The **add** method inserts a new element into the list immediately before the current cursor position.

A note on ListIterators

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

public interface **Queue<E>** extends Collection<E>

- **Queue** — a collection used to hold multiple elements prior to processing.
- Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

public interface **Queue<E>** extends Collection<E>

```
public interface Queue<E> extends Collection<E> {
    E element();                        //throws
    E peek();                      //null
    boolean offer(E e);  //add - bool
    E remove();                         //throws
    E poll();                      //null
}
```

public interface **Map<K,V>**

- **Map** — an object that maps keys to values.
- A Map cannot contain duplicate keys; each key can map to at most one value.
- Think about DSAHashTable - you're already familiar with the basics of Map.

public interface **Map<K,V>**

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
```

public interface **SortedSet<E>** extends Set<E>

- **SortedSet** — a Set that maintains its elements in ascending order.
- Several additional operations are provided to take advantage of the ordering.
- Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

public interface **SortedSet<E>** extends Set<E>

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Note on Comparator interface

– Comparator is another interface (in addition to Comparable) provided by the Java API which can be used to order objects.

– You can use this interface to define an order that is different from the Comparable (natural) order.

public interface **SortedMap<K,V>** extends Map<K,V>

- **SortedMap** — a Map that maintains its mappings in ascending key order.
- This is the Map analog of SortedSet.
- Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

public interface **SortedMap<K,V>** extends Map<K,V>

```
public interface SortedMap<K, V> extends Map<K, V>{

    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();

    Comparator<? super K> comparator();
}
```

General-purpose Implementations					
Interfaces	**Implementations**				
	Hash table	**Resizable array**	**Tree <u>(sorted)</u>**	**Linked list**	**Hash table + Linked list**
Set	**HashSet**		TreeSet **<u>(sorted)</u>**		LinkedHashSet
List		**ArrayList**		LinkedList	
Queue					
Map	**HashMap**		TreeMap **<u>(sorted)</u>**		LinkedHashMap

Note the naming convention

LinkedList also implements queue and there is a PriorityQueue implementation (implemented with heap)

Implementations

- Each of the implementations offers the strengths and weaknesses of the underlying data structure.
- What does that mean for:
  - Hashtable
  - Resizable array
  - Tree
  - LinkedList
  - Hashtable plus LinkedList
- **<u>Think about these tradeoffs when selecting the implementation!</u>**

Choosing the datatype

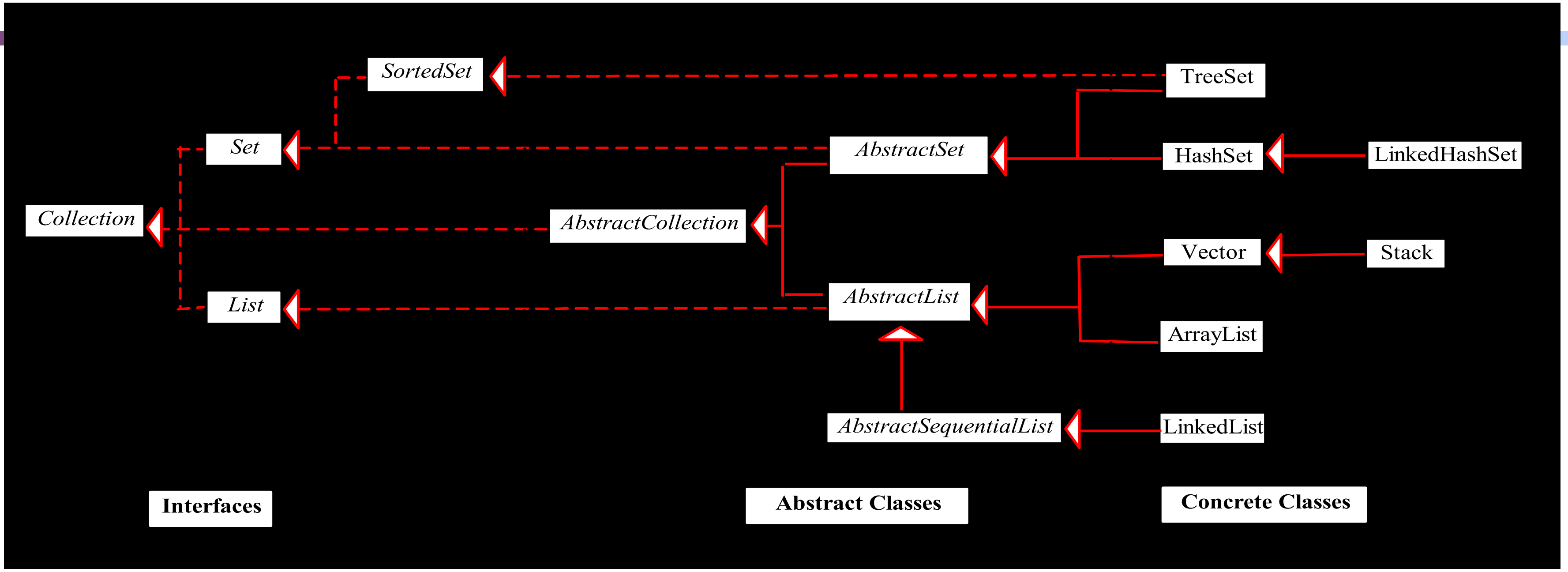
- When you declare a Set, List or Map, you should use Set, List or Map interface as the datatype instead of the implementing class.
- That will allow you to change the implementation by changing a single line of code!

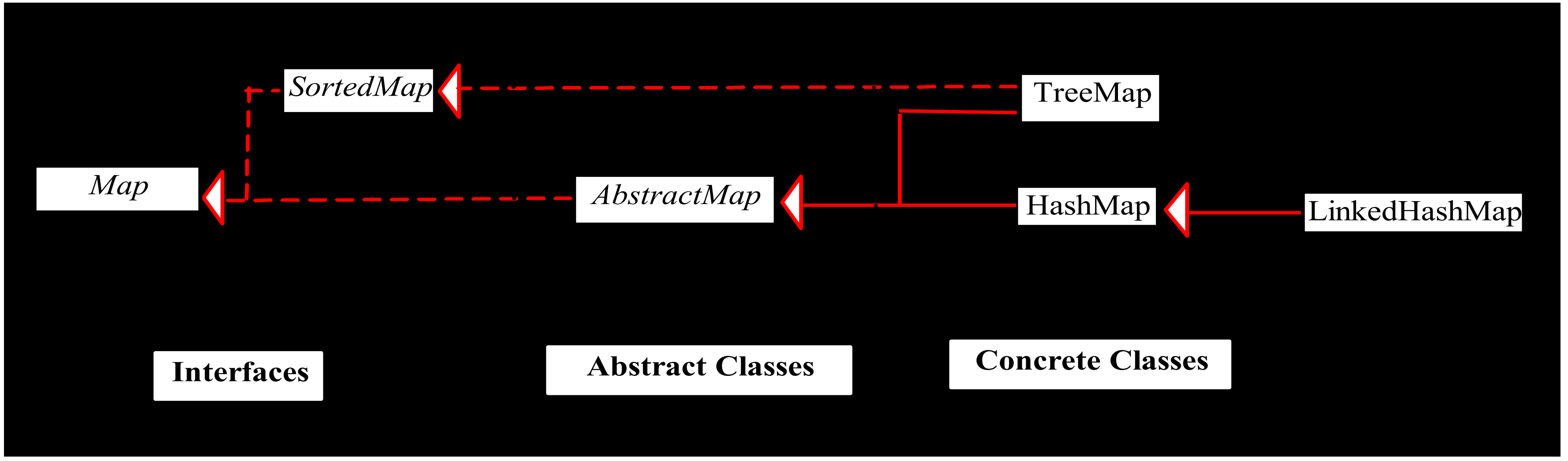
```
import java.util.*;

public class Test {
    public static void main(String[] args) {
        Set<String> ss = new LinkedHashSet<String>();

        for (int i = 0; i < args.length; i++)
            ss.add(args[i]);
        Iterator i = ss.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

```
import java.util.*;

public class Test {

    public static void main(String[] args)
    {
        //map to hold student grades
        Map<String, Integer> theMap = new HashMap<String, Integer>();

        theMap.put("Korth, Evan", 100);
        theMap.put("Plant, Robert", 90);
        theMap.put("Coyne, Wayne", 92);
        theMap.put("Franti, Michael", 98);
        theMap.put("Lennon, John", 88);

        System.out.println(theMap);
        System.out.println("--------------------------------------");
        System.out.println(theMap.get("Korth, Evan"));
        System.out.println(theMap.get("Franti, Michael"));
    }
}
```

Other implementations in the API

- Wrapper implementations delegate all their real work to a specified collection but add (or remove) extra functionality on top of what the collection offers.
  - Synchronization Wrappers
  - Unmodifiable Wrappers
- Convenience implementations are mini-implementations that can be more convenient and more efficient than general-purpose implementations when you don't need their full power
  - List View of an Array
  - Immutable Multiple-Copy List
  - Immutable Singleton Set
  - Empty Set, List, and Map Constants


SortedSet
TreeSet
Set
AbstractSet
HashSet
LinkedHashSet
Collection
AbstractCollection
Vector
Stack
List
AbstractList
ArrayList
AbstractSequentialList
LinkedList
Interfaces
Abstract Classes
Concrete Classes


SortedMap
TreeMap
Map
AbstractMap
HashMap
LinkedHashMap
Interfaces
Abstract Classes
Concrete Classes

Making your own implementations

- Most of the time you can use the implementations provided for you in the Java API.
- In case the existing implementations do not satisfy your needs, you can write your own by extending the abstract classes provided in the collections framework.

Algorithms

– The Java collections framework also provides polymorphic versions of algorithms you can run on collections...

- Sorting
- Shuffling
- Routine Data Manipulation
  - » Reverse
  - » Fill copy
  - » etc.
- Searching
  - » Binary Search
- Composition
  - » Frequency
  - » Disjoint
- Finding extreme values
  - » Min / Max

Python Built-Ins

- For DSA, we have specifically avoided built-ins that were covered in FOP
- We can now revisit these standard library options to see how they align with our DSA topics:
  - Sorting
  - Sets
  - Stacks and Queues
  - Heaps – priority queues
  - Hash Tables

https://dbader.org/blog/fundamental-data-structures-in-python

Python Built-ins: Sorting

- If you have a list, there is a sort method available
- Most types can be converted into a list, sorted and sent back
- Pandas and numpy provide sorting methods for dataframes and arrays
  - `sorted_list = old_list.sort( )`
  - `sorted(old_list)    # sorts the original list`
  - `rev_list = old_list.sort(reverse=True)`

Python Built-ins: Sets

- A *set* is an unordered collection of objects that does not allow duplicate elements.
- **Functionality**: test a value for **membership** in the set, **insert** or **delete** new values from a set, and to compute the **union** or **intersection** of two sets.
- In a "proper" set implementation, membership tests are expected to run in $O(1)$ time. Union, intersection, difference, and subset operations should be $O(n)$.
- The set implementations in Python match this performance.

Python Built-ins: Sets

– Examples of using sets in Python:

```
>>> vowels = {'a', 'e', 'i', 'o', 'u'}
>>> 'e' in vowels
True
>>> letters = set('alice')
>>> letters.intersection(vowels) {'a', 'e', 'i'}
>>> vowels.add('x')
>>> vowels {'i', 'a', 'u', 'o', 'x', 'e'}
>>> len(vowels)
6
>>> squares = {x * x for x in range(10)}
```

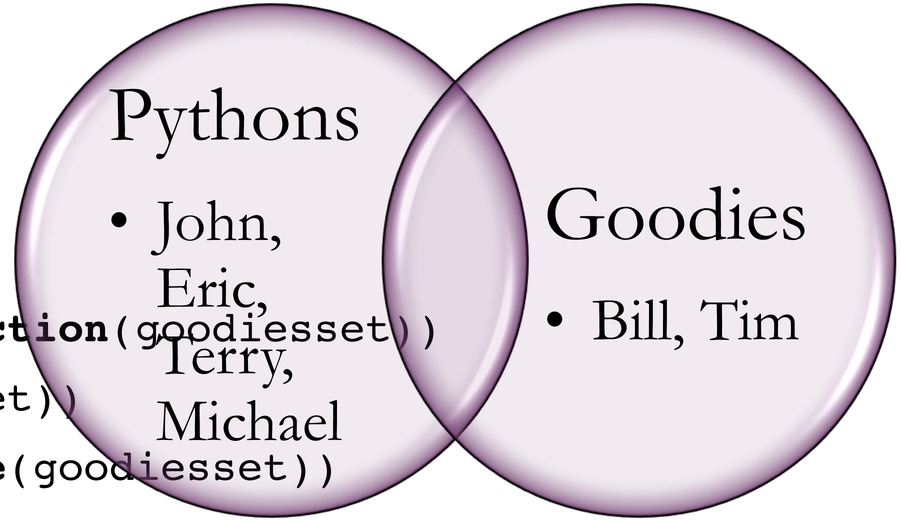
Set creation and operations (from FOP)

```
pythonlist = ['John', 'Eric', 'Graham', 'Terry', 'Michael', 'Terry']
pythonset = set(pythonlist)
goodieslist = ['Bill', 'Graham', 'Tim']
goodiesset = set(goodieslist)
print(pythonset)
print(goodiesset)
print('Intersection = ', pythonset.intersection(goodiesset))
print('Union = ', pythonset.union(goodiesset))
print('Difference = ', pythonset.difference(goodiesset))
print('Difference = ', goodiesset.difference(pythonset))
```

```
{'Eric', 'John', 'Michael', 'Terry', 'Graham'}
{'Tim', 'Bill', 'Graham'}
Intersection =  {'Graham'}
Union =  {'John', 'Michael', 'Tim', 'Bill', 'Eric', 'Terry', 'Graham'}
Difference =  {'Eric', 'John', 'Terry', 'Michael'}
Difference =  {'Bill', 'Tim'}
```

Set creation and operations (from FOP)

```
pythonlist = ['John', 'Eric', 'Graham', 'Terry', 'Michael', 'Terry']
pythonset = set(pythonlist)
goodieslist = ['Bill', 'Graham', 'Tim']
goodiesset = set(goodieslist)
print(pythonset)
print(goodiesset)
print('Intersection = ', pythonset.intersection(goodiesset))
print('Union = ', pythonset.union(goodiesset))
print('Difference = ', pythonset.difference(goodiesset))
print('Difference = ', goodiesset.difference(pythonset))
```




```
{'Eric', 'John', 'Michael', 'Terry', 'Graham'}
{'Tim', 'Bill', 'Graham'}
Intersection =  {'Graham'}
Union =  {'John', 'Michael', 'Tim', 'Bill', 'Eric', 'Terry', 'Graham'}
Difference =  {'Eric', 'John', 'Terry', 'Michael'}
Difference =  {'Bill', 'Tim'}
```

Python Built-ins: Multi-Sets

– Multiset (or bag) type allows multiple occurrences of elements in the set

```
>>> from collections import Counter
>>> inventory = Counter()
>>> loot = {'sword': 1, 'bread': 3}
>>> inventory.update(loot)
>>> inventory
Counter({'bread': 3, 'sword': 1})
>>> more_loot = {'sword': 1, 'apple': 1}
>>> inventory.update(more_loot)
>>> inventory
Counter({'bread': 3, 'sword': 2, 'apple': 1})
```

Python Built-ins: Queues

– Could use a List to provide queue behaviour, but the performance is poor:

```
# How to use Python's list as a FIFO queue:
q = []
q.append('eat')
q.append('sleep')
q.append('code')
>>> q
['eat', 'sleep', 'code']
# Careful: This is slow!
>>> q.pop(0)
'eat'
```

Python Built-ins: Queues (deque)

– The deque class implements a double-ended queue that supports adding and removing elements from either end in *O(1)* time.

– deque objects are implemented as doubly-linked lists

  - excellent performance for enqueuing and dequeuing
  - poor *O(n)* performance for randomly accessing elements in the middle of the queue.

– Because deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.

Python Built-ins: Queues (deque)

```
# How to use collections.deque as a FIFO queue:
from collections import deque
q = deque()
q.append('eat')
q.append('sleep')
q.append('code')
>>> q                          deque(['eat', 'sleep', 'code'])
>>> q.popleft()                'eat'
>>> q.popleft()                'sleep'
>>> q.popleft()                'code'
>>> q.popleft()
IndexError: "pop from an empty deque"
```

Python Built-ins: Priority Queue (Heap)

- Priority queue is a modified queue:
  - instead of retrieving by insertion time, retrieve by *highest-priority*
- The priority of individual elements is decided by the ordering applied to their keys.

```
from queue import PriorityQueue
q = PriorityQueue()
q.put((2, 'code'))
q.put((1, 'eat'))
q.put((3, 'sleep'))
while not q.empty():
    next_item = q.get()
    print(next_item)
```

Result:

```
(1, 'eat')
(2, 'code')
(3, 'sleep')
```

Python Built-ins: Dictionary (Hashtable)

- The dictionary abstract data type is one of the most frequently used and most important data structures in computer science.
- Because of this importance Python features a robust dictionary implementation as one of its built-in data types (dict).
- Python's dictionaries are indexed by keys of any hashable type.
- A hashable object has a hash value which never changes in its lifetime (__hash__), and can be compared (__eq__).

```
>>> phonebook = {'bob': 7387, 'alice': 3719, 'jack': 7052}
>>> phonebook['alice']
3719
```

Python Built-ins: Dictionary (Hashtable)

– Python dictionaries are based on a well-tested and finely tuned hash table implementation that provides the performance characteristics you'd expect:
  - *O(1)* time complexity for lookup, insert, update, and delete operations in the average case.

– Can map to objects for hash table entries:

```
>>> items = {'AAA' : [4, 5, 6, 7],
    'BBB' : [10,20,30,40], 'CCC': [100,50,-30,-50]}
>>> items['AAA']
[4, 5, 6, 7]
```

Dictionary – The Meaning of Liff

```
liff = {'Duddo': 'The most deformed potato in any given
collection of potatoes.',
        'Fring': 'The noise made by a lightbulb that has
just shone its last.',
        'Tonypandy': ' The voice used by presenters on
children\'s television programmes.'}
liff['Wawne'] = 'A badly supressed yawn.'
liff['Woking'] = 'Standing in the kitchen wondering what you
came in here for.'
print(liff)
print(liff['Duddo'])
print(liff['Fring'])
print(liff.keys())
del liff['Fring']
print(liff.keys())
```

The Meaning of Liff and
The Deeper Meaning of Liff,
by Douglas Adams and John Lloyd

http://liff.hivemind.net/

Dictionary – The Meaning of Liff

```
OUTPUT
{'Fring': 'The noise made by a lightbulb that has just shone
its last.', 'Wawne': 'A badly supressed yawn.', 'Duddo': 'The
most deformed potato in any given collection of potatoes.',
'Tonypandy': " The voice used by presenters on children's
television programmes.", 'Woking': 'Standing in the kitchen
wondering what you came in here for.'}

The most deformed potato in any given collection of potatoes.

The noise made by a lightbulb that has just shone its last.
dict_keys(['Woking', 'Fring', 'Wawne', 'Tonypandy', 'Duddo'])
dict_keys(['Woking', 'Wawne', 'Tonypandy', 'Duddo'])
```

Dictionaries

– Maps that create a set of relationships between keys and values.

```
pops = {'New South Wales': 7757843,
        'Victoria' : 6100877,
        'Queensland' : 4860448,
        'South Australia' : 1710804,
        'Western Australia' : 2623164,
        'Tasmania': 519783,
        'Northern Territory' : 245657,
        'Australian Capital Territory': 398349}

print(pops['Victoria'])          6100877
print(pops['Queensland'])        4860448
```

Australian Demographic Statistics, Sep 2016
http://www.abs.gov.au/ausstats/abs@.nsf/mf/3101.0

Dictionaries

– We can list the keys, or the values, or both...

```
for p in pops:
    print(p)
```

```
Tasmania
Western Australia
…
Australian Capital Territory
New South Wales
```

```
for k in pops.keys():
    print(pops[k])
```

```
519783
2623164
...
7757843
```

```
for k in pops.keys():
    print(k, ': ', pops[k])
```

```
Tasmania:  519783
Western Australia:  2623164
…
New South Wales:  7757843
```


71

Dictionary: Word Frequencies…

– Find the frequency of each of the words in a text…

```
import sys

punctuation = '~!@#$%^&*()_+{}|:"<>?`=[]\\;\',./'

if len(sys.argv) <2 :
    filename = 'grimm.txt'
else:
    filename = sys.argv[1]

book = open(filename).read()
bookP = book.translate(str.maketrans('','',punctuation))
words = bookP.lower().split()
print(len(words))
print(words[:10])
```

```
1139
['rumpelstiltskin', 'by',
'the', 'side', 'of', 'a',
'wood', 'in', 'a', 'country']
```

Dictionary: Word Frequencies…

– Then calculate frequencies using a dictionary…

```
wordfreq = {}                     # empty dictionary
for word in words:                # for each word
    if word not in wordfreq:      # if it's not in dict
        wordfreq[word] = 0        # create a key/val pair
    wordfreq[word] += 1           # increment count[word]

print(len(wordfreq))              # 390 unique, 1139 total
print(wordfreq)
```

– There are many alternative packages with extensive support for analysing text (e.g. nltk) – but this is a good starting point

Conclusion: What now?

- For the exam, you should not use Built-ins
- Beyond that, explore them, compare them, use them
- Now that you get ADTs and algorithms, you can use them (or not) from a point of understanding
- You should be confident that you can research and select algorithms and ADTs, and beyond that APIs
- You can then write code to work with and extend on the work of others