Structured Data in Java
The Collections Framework

Implementations

- In the previous lecture, we've discussed interfaces
  - They specify method headers but not method implementations
- To actually use these interfaces, we need concrete classes that implement them
  - These concrete classes will still be generic
  - But they will provide actual code to store the data in some way that obeys the specification of the interface
- Many possible ways to implement any given interface
  - Each will have its advantages and disadvantages

General-purpose Implementations

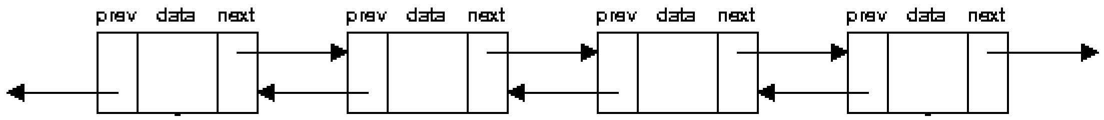
Interfaces	**Implementations**				
	Hash table	Resizable array	Binary Search tree	Linked list	Hash table + Linked list
Set	**HashSet**		**TreeSet** (sorted)		**LinkedHashSet**
List		**ArrayList**		**LinkedList**	
Map	**HashMap**		**TreeMap** (sorted)		**LinkedHashMap**

- Each of the implementations offers the strengths and weaknesses of the underlying data structure.
- **Think** about these tradeoffs when selecting the implementation!

Resizable Arrays vs Linked Lists

- First we need to understand a resizable array
  - Once an array is allocated, its size cannot change
  - So to resize an array, you need to allocate a completely new array and copy all the data from the original array

- Next we need to understand a linked list
  - Rather than storing elements in adjacent memory locations, a linked list stores elements in individual packages (nodes) where each node has links to its predecessor and successor in the list

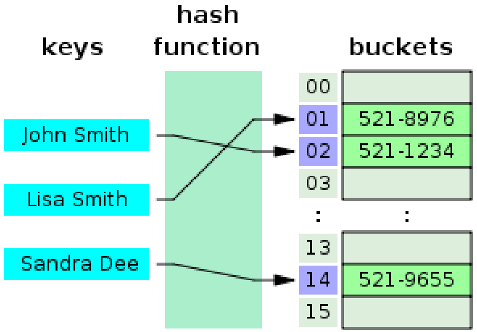


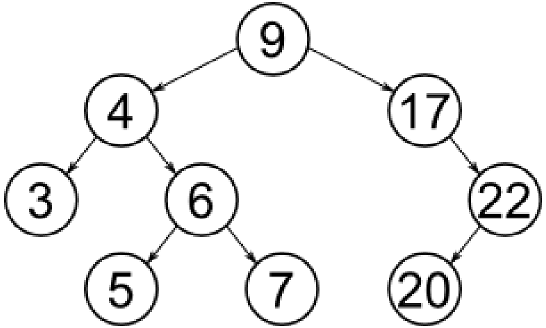
Resizable Arrays vs Linked Lists

- Indexing:
  - Arrays give us direct access to any element given its index, O(1)
  - Linked lists must traverse the list to get to a specific index, O(n)
- Insertion/deletion:
  - Arrays may have to shift all other elements up or down, O(n)
  - Linked lists can easily link/unlink a node in the list, O(1)
  - Arrays may have to resize, O(n), whereas linked lists do not
- Memory:
  - Arrays only hold the elements being stored
  - Nodes in linked list contain the element plus links

Hash Table vs. Binary Search Tree

- First we need to understand a hash table
  - Employs a function to distribute elements randomly over a table
  - Need to handle collisions when two elements map to the same table entry
- Next we need to understand a binary search tree
  - Elements stored in a tree fashion that maintains parent/child links
  - Maintains sorted order to allow fast searches





Hash Table vs. Binary Search Tree

- Insertion/Lookup
  - Hash table gives us immediate access to a pair via the hash function, O(1)
  - Binary search tree must be walked, O(lg n)

- Processing all elements
  - Binary search trees allow us to process all elements in sorted order
  - Hash tables are unordered

Choosing the datatype

- When you declare any type of 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!

```
Set<String> ss = new LinkedHashSet<String>();
```

```
LinkedHashSet<String> ss = new LinkedHashSet<String>();
```

Algorithms

- The collections framework also provides polymorphic versions of algorithms you can run on collections
  - Sorting
  - Searching
    - Binary search
  - Shuffling/rotating
  - Routine data manipulation
    - Reverse
    - Fill copy
  - Finding extreme values
    - Min
    - Max

Iterating Over Collections

- Once we have a bunch of element stored in a collection, we often want to process them all
  - We call this iterating over the collection

- There are several ways to do this in Java, we will discuss three:
  1. Using indexing for collections that allow it
  2. Using a for-each loop
  3. Using an iterator

Iterating Over Collections

- If the collection provides methods that allow indexing, we can process all the elements with a conventional `for` loop

```
List<String> myFriends = new ArrayList<String>();
myFriends.add("Pete");
myFriends.add("Lisa");
myFriends.add("Gus");
for (int i=0; i<myFriends.size(); i++) {
   out.println(myFriends.get(i));
}
```

- This works, but it is not the preferred way

Iterating Over Collections

- The `for-each` loop that we saw with arrays can also be used to process the elements of a collection

```
List<String> myFriends = new ArrayList<String>();
myFriends.add("Pete");
myFriends.add("Lisa");
myFriends.add("Gus");
for (String aFriend : myFriends) {
   out.println(aFriend);
}
```

- This works very well, and is clean & concise

Iterating Over Collections

- We can also use what is known as an iterator, which is a generic mechanism for iterating over a generic collection
- Every collection has an `iterator()` method that returns an `Iterator` object over its elements
- An `Iterator` has the following methods:

`hasNext()`	returns `true` if there are more elements to examine
`next()`	returns the next element from the collection
`remove()`	removes the last value returned by `next()`

Iterating Over Collections

```
List<String> myFriends = new ArrayList<String>();
myFriends.add("Pete");
myFriends.add("Lisa");
myFriends.add("Gus");

Iterator<String> itr = myFriends.iterator();
while (itr.hasNext()) {
   String aFriend = itr.next();
   out.println(aFriend);
}
```

- This works very well and allows deletion, but maybe a bit verbose

Iterators

- Iterators provide a generic way to traverse through a collection regardless of its implementation – works for arrays, linked list, binary trees, hash tables, etc.
- Sometimes we don't know the exact implementation of Iterator we are getting, but **we don't care**, as long as it provides the methods `next()` and `hasNext()`
- Good practice: ***Program to an interface!***

Iterators and Maps

- The `Map` interface does not provide for an iterator
- But a `Map` has two methods:
  - `Set<K> keySet()`
    - Returns a set view of the keys contained in this map
  - `Collection<V> values()`
    - Returns a collection view of the values contained in this map
- And you can iterate over these collections