CS 600.226: Data Structures

Michael Schatz

Oct 22, 2018

Lecture 22. Ordered Sets

HW5

Assignment 5: Six Degrees of Awesome

Out on: October 17, 2018

Due by: October 26, 2018 before 10:00 pm

Collaboration: None

Grading:

- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

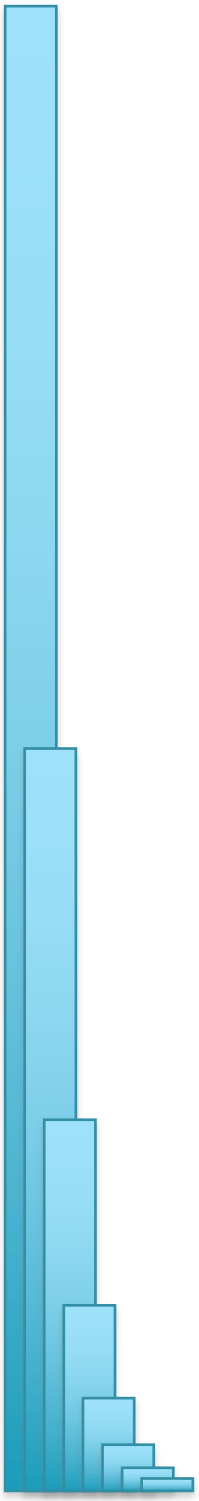
Overview

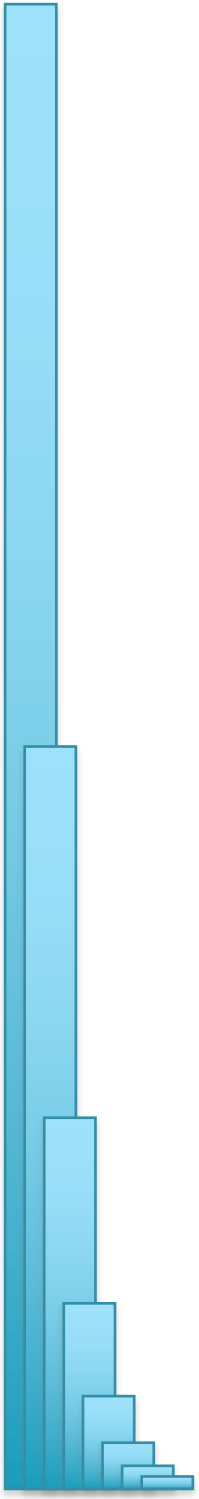
The fifth assignment is all about graphs, specifically about graphs of movies and the actors and actresses who star in them. You'll implement a graph data structure following the interface we designed in lecture, and you'll implement it using the incidence list representation.

Turns out that this representation is way more memory-efficient for sparse graphs, something we'll need below. You'll then use your graph implementation to help you play a variant of the famous Six Degrees of Kevin Bacon game. Which variant? See below!

Agenda

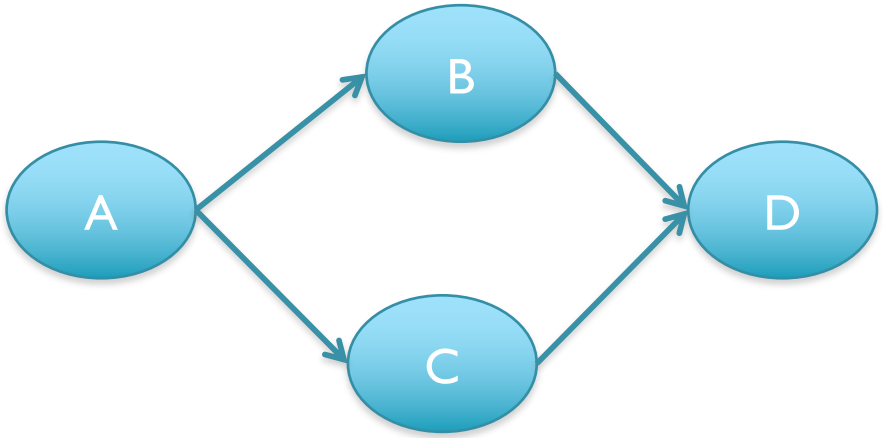
1. ***Recap on Sets***
2. ***Ordered Sets***



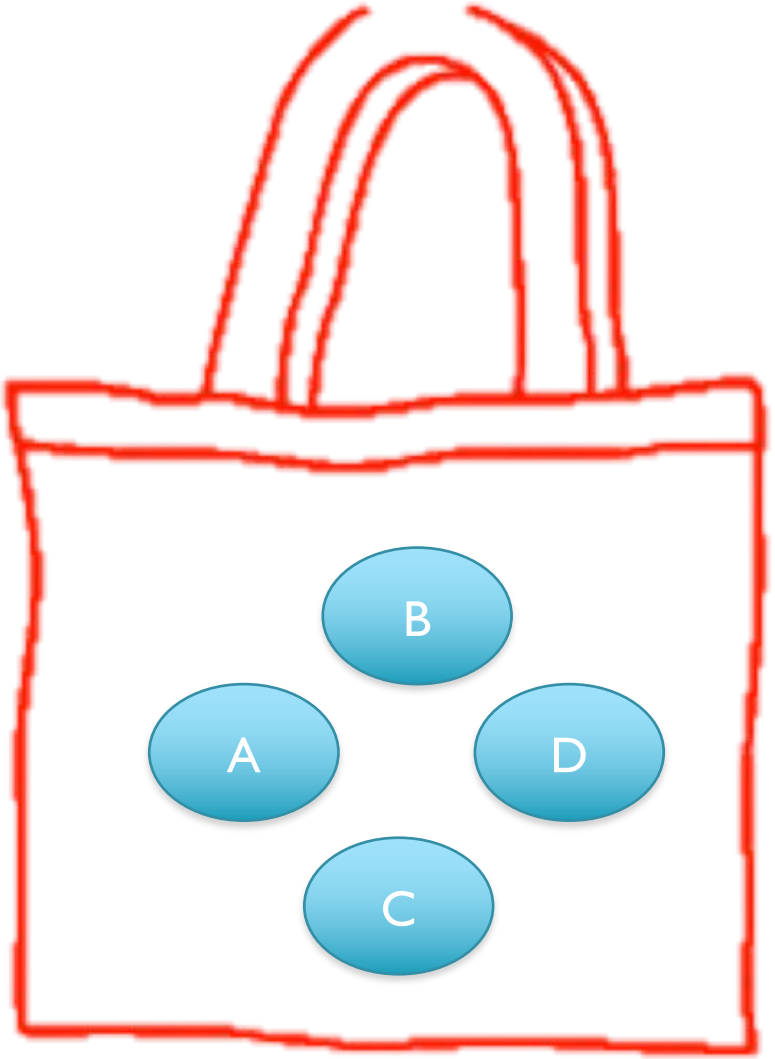


Part 1:Sets

Graphs versus Sets



Position-Based



Value-Based

Set Interface

```
public interface Set<T> implements Iterable<T> {
    void insert(T t);
    void remove(T t);
    boolean has(T t);

    boolean empty();
    T any() throws EmptySetException;

    Iterator<T> iterator();
}
```

Now we can actually get all the values without destroying the set ☺

Implementations

Ideas????

(Doubly) Linked List: Fast add/remove once found, slow to find

Array: Fast add, remove can be slow, still have to find

Tree: Fast add, maybe fast to remove?, still have to find, keep tree balanced

Stack, Queue, Deque, Graph: Probably wont work ☹

We can make add O(1) by "cheating" and allowing multiple copies and delete them all on remove … but requires O(n) space!

Speed up Array by marking elements deleted but don't shift things down (allow gaps in array)

Sorting may be really useful so can do lookup in lg(n) time ☺ … but only if the type supports a comparison!

ArraySet

```
private int find(T t) {
    for (int i = 0; i < this.length; i++) {
        if (this.data[i].equals(t)) { return i; }
    }
    return -1;
}

public void remove(T t) {
    int position = this.find(t);
    if (position == -1) {return; }
    for (int i = position; i < this.length -1; i++) {
        this.data[i] = this.data[i+1];
    }
    this.length -= 1;
}

public boolean has(T t) {
    return this.find(t) != -1;
}

public void insert(T t) {
    if (this.has(t)) { return; }
    if (this.length == this.data.length) { this.grow(); }
    this.data[this.length] = t;
    this.length += 1;
}
```

Performance Testing

```
$ seq 1 1000    | awk '{print int(rand()*100000)}' > random1k.txt
$ seq 1 10000   | awk '{print int(rand()*100000)}' > random10k.txt
$ seq 1 100000  | awk '{print int(rand()*100000)}' > random100k.txt

$ time java Unique < random1k.txt > bla
real 0m0.176s
user 0m0.242s
sys  0m0.041s

$ wc –l bla
993 bla

$ time java Unique < random10k.txt > bla
real 0m0.368s
user 0m0.786s
sys  0m0.088s

$ wc –l bla
9529 bla

$ time java Unique < random100k.txt > bla
real 0m4.466s
user 0m4.735s
sys  0m0.279s

$ wc –l bla
63110 bla
```

Where does the time go?

```
$ java -agentlib:hprof=cpu=times UniqueList < random1k.txt > bla
$ less java.hprof.txt

…

CPU TIME (ms) BEGIN (total = 3556) Wed Oct 19 23:51:10 2016
rank   self  accum   count trace method
   1 30.34% 30.34%  494953 304975 java.lang.Integer.equals
   2 16.73% 47.08%    1000 304958 ListSet.find
   3 12.09% 59.17%  494953 304974 java.lang.Integer.intValue
   4  0.96% 60.12%    8918 304826 java.nio.HeapCharBuffer.get
   5  0.87% 61.00%    5918 304837 java.util.regex.Pattern$CharProperty.match
   6  0.84% 61.84%    1988 305038 sun.nio.cs.UTF_8$Encoder.encodeArrayLoop
   7  0.82% 62.65%    6921 304827 java.nio.CharBuffer.charAt
   8  0.79% 63.44%    1008 304796 java.util.Scanner.getCompleteTokenInBuffer
   9  0.73% 64.17%    5918 304835 java.lang.Character.codePointAt
  10  0.59% 64.76%    4903 304874 java.util.regex.Pattern$BmpCharProperty.match
  11  0.56% 65.33%    1988 305064 sun.nio.cs.StreamEncoder.writeBytes
  12  0.56% 65.89%    6921 304833 java.lang.Character.isWhitespace
  13  0.53% 66.42%    1988 305045 sun.nio.cs.StreamEncoder.implWrite
  14  0.51% 66.93%    4903 304943 java.lang.Character.digit
  15  0.51% 67.44%    8918 304832 java.lang.CharacterDataLatin1.isWhitespace
  16  0.45% 67.89%    1988 305036 sun.nio.cs.UTF_8.updatePositions
  17  0.45% 68.34%    4903 304865 java.nio.HeapCharBuffer.get
  18  0.45% 68.79%    3000 304908 java.nio.HeapCharBuffer.subSequence
  19  0.42% 69.21%    4903 304866 java.nio.CharBuffer.charAt
  20  0.42% 69.63%    5918 304838 java.util.regex.Pattern$Curly.match
```

What memory did we use?

```
$ java -agentlib:hprof UniqueList < random1k.txt > bla
$ less java.hprof.txt

…

SITES BEGIN (ordered by live bytes) Wed Oct 19 23:58:28 2016
          percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs      bytes  objs trace name
    1 14.47% 14.47%    304000 1000     304000  1000 301198 int[]
    2  8.00% 22.47%    168000 1000     168000  1000 301199 int[]
    3  6.86% 29.32%    144000 3000     144000  3000 301200 java.nio.HeapCharBuffer
    4  5.29% 34.61%    111024 1413     111024  1413 300000 char[]
    5  4.61% 39.22%     96768 1008      96768  1008 301195 int[]
    6  4.57% 43.78%     95952 1999      95952  1999 300265 java.nio.HeapCharBuffer
    7  4.47% 48.26%     93960 3000      93960  3000 301202 char[]
    8  3.43% 51.68%     72000 3000      72000  3000 301201 java.lang.String
    9  1.94% 53.62%     40680  172      40680   172 300011 char[]
   10  1.62% 55.24%     34096 1399      34096  1399 300000 java.lang.String
   11  1.52% 56.77%     32024  554      32024   554 300000 java.lang.Object[]
   12  1.48% 58.25%     31096  993      31096   993 301207 char[]
   13  1.21% 59.46%     25504    8      25504     8 300000 byte[]
   14  1.17% 60.63%     24624    3      24624     3 300259 byte[]
   15  1.16% 61.80%     24448  326      24448   326 301241 int[]
   16  1.13% 62.93%     23832  993      23832   993 301206 java.lang.String
   17  1.13% 64.07%     23832  993      23832   993 301205 ListSet$Node
   18  0.78% 64.85%     16400    1      16400     1 300674 char[]
   19  0.78% 65.63%     16400    1      16400     1 300262 char[]
   20  0.77% 66.40%     16128 1008      16128  1008 301196 int[]
```

jb: Simple Go-style benchmarking for Java

Welcome to `jb`, also known as `jaybee` out in the shell.

This is a port of Go's benchmarking facilities to Java. I needed `jb` for the data structures course I taught in Fall 2016, mostly to make it easier on students to get some basic performance metrics for their code. The tool has been reasonably well-received over the years (Spring 2017, Fall 2017, Spring 2018) and students who have previously learned about JUnit 4 seem to pick it up with ease.

I borrowed *very* liberally from the Go original as well as from the ports listed below. One might say that my only accomplishment was to package things in a slightly more Java-like way. (Or maybe the only real advantage is that I also remixed a cute logo for it? Bzzz! Oh, I would like to apologize for taking the whole bee metaphor a little too far in the source code.)

I welcome pull requests!

Building

If you want to build `jaybee` yourself, you'll first need to install maven. I actually feel a bit guilty about having switched from a hacky `Makefile` to an insanely huge monster of a build system, but hey, it's 2018 so let's waste lots of cycles and even more space to do very simple things.

Once you have maven installed go ahead and clone the repository. Then say `mvn install` and after waiting patiently for a while you'll have the JAR file referenced below.

If you are on a UNIX system, you should be able to say `make` which will run `mvn install` as above before packaging the JAR into a "fake executable" called `jaybee` that can be used to run benchmarks more conveniently. (At some point I might add `jaybeecee` as well, let's see.)

Usage

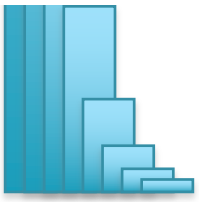
If `StringAppend.java` contains a bunch of methods annotated with `@Bench`, here's how you compile it (provided `jaybee.jar` is in the current directory):

```
$ javac -classpath jaybee.jar:. StringAppend.java
```

If `StringAppend.class` contains a bunch of compiled benchmark methods, here's how you run them (provided `jaybee.jar` is in the current directory):

```
$ java -jar jaybee.jar StringAppend
       string          5,000        326,858 ns/op          8.84 MB/s          156,993 B/op
  stringBuffer         50,000         26,474 ns/op        109.16 MB/s           18,685 B/op
stringBuilder        100,000         13,622 ns/op        212.15 MB/s              339 B/op
```

You get (a) the number of times the benchmark was run, (b) the time it took per iteration, (c) the amount of data processed per second, and (d) the number of bytes allocated per iteration. (That last number can be rather flakey because garbage collection behavior is not exactly deterministic.)



Webpage: https://github.com/phf/jb
Available in resources/jaybee-1.0.jar

CompareSets.java

```
import com.github.phf.jb.Bench;
import com.github.phf.jb.Bee;
import java.util.Random;

public final class CompareSets {
    private static final Random r = new Random();
    private static final int SPREAD = 100000;
    private static final int SIZE   = 1000;

    @Bench
    public void randomArraySet(Bee b) {
      for (int i = 0; i < b.reps(); i++)
      {
        ArraySet<Integer> set = new ArraySet<Integer>();
        for (int j = 0; j < SIZE; j++)
        {
          set.insert(r.nextInt(SPREAD));
        }
        b.bytes(SIZE * 4);
      }
    }

    @Bench
    public void randomListSet(Bee b) {
      for (int i = 0; i < b.reps(); i++)
      {
        ListSet<Integer> set = new ListSet<Integer>();
        for (int j = 0; j < SIZE; j++)
        {
          set.insert(r.nextInt(SPREAD));
        }
        b.bytes(SIZE * 4);
      }
    }
}
```

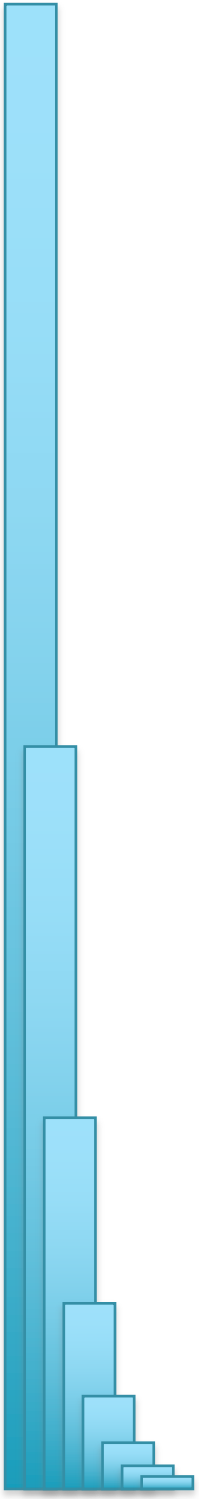
Just like Junit!
@Bench

Just like Junit!
@Bench

CompareSets.java

```
import com.github.phf.jb.Bench;
import com.github.phf.jb.Bee;
import java.util.Random;

public final class CompareSets {
    private static final Random r = new Random();
    private static final int SPREAD = 100000;
    private static final int SIZE   = 1000;

    @Bench
    public void randomArraySet(Bee b) {
      for (int i = 0; i < b.reps(); i++)
      {
        ArraySet<Integer> set = new ArraySet<Integer>();
        for (int j = 0; j < SIZE; j++)
        {
          set.insert(r.nextInt(SPREAD));
        }
        b.bytes(SIZE * 4);
      }
    }

    public void randomListSet(Bee b) {
      for (int i = 0; i < b.reps(); i++)
      {
        ListSet<Integer> set = new ListSet<Integer>();
        for (int j = 0; j < SIZE; j++)
        {
          set.insert(r.nextInt(SPREAD));
        }
        b.bytes(SIZE * 4);
      }
    }
}
```

```
$ javac -cp .:jb/bin/jaybee.jar CompareSets.java
$ java –jar jaybee.jar CompareSets

randomArraySet   3000     434119 ns/op      9.21 MB/s      1839 B/op
 randomListSet   1000    1270848 ns/op      3.15 MB/s      6190 B/op
```



Part 1.5: Self-Organizing Sets

Can we make these go faster?

Consider the input:

```
1 2 3 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 …
```

Can we change has() to speed it up?

```
class ArraySet
 private int find(T t) {
    for (int i = 0; i < this.length; i++) {
      if (this.data[i].equals(t)) {
        return i;
      }
    }
    return -1;
  }
  public boolean has(T t) {
    return this.find(t) != -1;
  }
  public void insert(T t) {
    if (this.has(t)) { return; }
  …
```

```
class ListSet
  private Node<T> find(T t) {
    for (Node<T> n = this.head;
         n != null;
         n = n.next) {
      if (n.data.equals(t)) { return n; }
    }
    return null;
  }
  public boolean has(T t) {
    return this.find(t) != null;
  }
  public void insert (T t) {
    if (this.has(t)) { return; }
  …
```

Can we change the internal array/list to speed it up?

Yes ☺

```
5 1 2 3 4
```

<= Will be about 5x faster for (very) long runs of 5

Performance Heuristics

Heuristic:

- Any approach to problem solving, learning, or discovery that employs a **practical method**, **not guaranteed to be optimal**, perfect, logical, or rational, but instead **sufficient** for reaching an immediate goal. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution.

Move-to-front Heuristic:

- If we are asked for find X and we do actually find it, we move that element to the front of the array or list so that it can be more quickly found next time

- **Example:**
  - If we start with [1 2 3 ... X ...],
  - After find(X) we shift the data to be [X 1 2 3 ...] instead.
  - If we are asked for 3 next, we'd shift to: [3 X 1 2 ...]
  - if we're asked for 3 again, nothing changes;
  - if we're asked for X again, we modify the list again: [X 3 1 2 ...]

Performance Heuristics

Move-to-front Heuristic:

- If we are asked for find X and we do actually find it, we move that element to the front of the array or list so that it can be more quickly found next time

- **Example:**
  - If we start with [1 2 3 ... X ...],
  - After find(X) we shift the data to be [X 1 2 3 ...] instead.

Locality of reference:

- If we're asked for element X, it's likely that we'll be asked for X again in the near future.
- Is this a law of nature? No, but works really well in practice

If we do this long enough, the order of elements in the list will reflect roughly how often they've been requested in the past. In other words, values that are "more popular" will take less time to find than values that are "less popular".

One can show that the performance of the move-to-front heuristic is never worse than two times the performance of the optimal list ordering

Performance Heuristics

Move-to-front Heuristic:

- If we are asked for find X and we do actually find it, we move that element to the front of the array or list so that it can be more quickly found next time

- **Example:**
  - If we start with [1 2 3 ... X ...],
  - After find(X) we shift the data to be [X 1 2 3 ...] instead.

How do you implement move-to-front on a ListSet()?

find() checks the data and if it matches the user data, stores a reference to that node in a new variable, and temporarily remove from the list. Then insert that node as the new beginning of the list.

What is the new complexity of find()?

Walk the monkey bars in O(n) to find the node, move to front in O(1) ☺

Any other issues?

ListSet iterator becomes very complex, don't do it :-(

Performance Heuristics

Move-to-front Heuristic:

- If we are asked for find X and we do actually find it, we move that element to the front of the array or list so that it can be more quickly found next time

- **Example:**
  - If we start with [1 2 3 ... X ...],
  - After find(X) we shift the data to be [X 1 2 3 ...] instead.

How do you implement move-to-front on a ArraySet()?

find() checks the data in the node. If node has the user data do what?

Swapping to the front wont work because on the next round it may get swapped back to the end

Sliding to the front works correctly, but doubles the runtime for find() :-(

Any other ideas?

Like bubblesort, on a successful find() we can shift it forward by one slot

This is called a transpose

Performance Heuristics

Transpose Heuristic:

- If we are asked for find X and we do actually find it, we move that element up one closer to the front

- **Example:**
  - Start: [1 2 3 4 5 6]
  - Asked for 5, we swap 4 and 5: [1 2 3 5 4 6].
  - Asked for 5 again: [1 2 5 3 4 6].
  - Asked for 2: [2 1 5 3 4 6]

Over time, values that are “more popular” will take less time to find than values that are “less popular”.

If nothing is ”popular” (random ordering), then the order will more-or-less reflect the order they were seen. Essentially no better but no worse than the original implementation

Performance Heuristics

Transpose Heuristic:

- If we are asked for find X and we do actually find it, we move that element up one closer to the front

- **Example:**
  - Start: [1 2 3 4 5 6]
  - Asked for 5, we swap 4 and 5: [1 2 3 5 4 6].
  - Asked for 5 aga
  - Asked for 2: [2 1

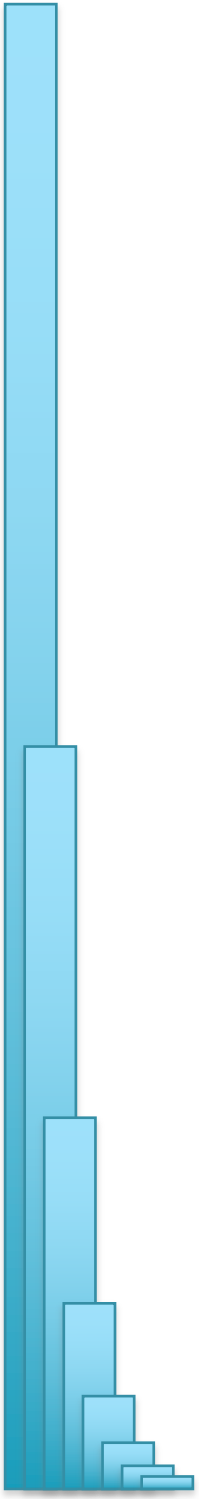
Over time, values that are “more popular” will take less time to find than values that are “less popular”.

```
private int find(T t) {
  for (int i = 0; i < length; i++) {
    if (this.data[i].equals(t)) {
      if (i > 0) {
        T x = this.data[i];
        this.data[i] = this.data[i-1];
        this.data[i-1] = x;
        return i-1;
      }
    return i;
    }
  }
  return -1;
}
```

UniqueArray vs UniqueTranspose

```
## Input the numbers 1 through 100,000
## Then 100,000 copies of 100,000

$ time ((seq 1 100000; jot -b 100000 100000)
 | java Unique > /dev/null)

real    0m36.439s
user    0m37.131s
sys     0m0.410s
```

```
$ time ((seq 1 100000; jot -b 100000 100000)
 | java UniqueTranspose > /dev/null)

real    0m25.987s
user    0m26.785s
sys     0m0.381s
```



Part 2: Ordered Sets

Set Interface

```
public interface Set<T> implements Iterable<T> {
    void insert(T t);
    void remove(T t);
    boolean has(T t);
}
```

With the Set inferface, we can make no assumptions about the methods available for type T. The only method we use is .equals() which is part of the java Object() class.

```
private int find(T t) {
    for (int i = 0; i < this.length; i++) {
      if (this.data[i].equals(t)) {
        return i;
      }
    }
    return -1;
  }
```

Wouldn't it be great if we could compare the objects….

Comparable

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

compact1, compact2, compact3
java.lang

Interface Comparable<T>

Type Parameters:

`T - the type of objects that this object may be compared to`

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
`int`	`compareTo(T o)` Compares this object with the specified object for order.

Method Detail

compareTo

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all x and y. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all z.

It is strongly recommended, but not strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(expression)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters:

`o - the object to be compared.`

Returns:

`a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.`

Throws:

`NullPointerException - if the specified object is null`

`ClassCastException - if the specified object's type prevents it from being compared to this object.`

Set Interface

```
public interface Set<T> implements Iterable<T> {
    void insert(T t);
    void remove(T t);
    boolean has(T t);
}
```

Only operation is if t1.equals(t2)

Cannot make a faster Set other than some performance heuristics

```
public interface OrderedSet <T extends Comparable<T>>
                                extends Iterable<T> {
    void insert(T t);
    void remove(T t);
    boolean has(T t);
}
```

We can compare if t1 < t2, t1 = t2, or t1 > t2

We can make OrderedSets go much faster! (how)?

Use binary search type techniques

Java Generics

Java will gladly let you make generic arrays casted from Object[1]

```
public ArraySet() {
    this.data = (T[]) new Object[1];
}
```

But will not let you do this for Comparable types!

```
public OrderedArraySet() {
    this.data = (T[]) new Comparable[1];
}
```

ERROR! Comparable is an interface!

- Instead of implementing our set using plain Java arrays we'll have to use some existing class that behaves mostly like an array.
- We could use our own, but Java already comes with a pretty good ArrayList<T> class, so let's use that.
- Note that despite the name ArrayList the class behaves more like an array and less like a linked list; the get operation, for example, is constant time.

ArrayListSet

```
import java.util.Iterator;
import java.util.ArrayList;

public class ArrayListSet <T> implements Set<T> {
    private ArrayList<T> data;

    public ArrayListSet() {
        this.data = new ArrayList<T>();
    }

    public void insert (T t) {
        if (this.has(t)) { return; }
        this.data.add(t);
    }

    private int find(T t) {
        for (int i = 0; i < this.data.size (); i++) {
            if (this.data.get(i).equals(t)) {
                return i;
            }
        }
        return -1;
    }
…
```

Use the ArrayList.add() that doubles for us

ArrayListSet

```
…

    public void remove(T t) {
        int position = this.find(t);
        if ( position == -1) { return; }
        this.data.remove(position);
    }

    public boolean has(T t) {
          return this.find(t) != -1;
    }

    public Iterator <T> iterator () {
        return this.data.iterator();
    }
}
```

Use the ArrayList.remove() that cleans up for us

We even get the iterator ☺

Be careful when…

… removing when iterating

Notice extensive use of find()

OrderedArrayListSet

Lets extend ArrayListSet to an OrderedArrayListSet. find() will always return the correct index for the value, regardless of whether the value is in the set or not

What is the "correct index" for a value? Here are the possible cases:

1. The set was ***empty*** before this insertion. The "correct index" for the value is ***0*** in this case, the first spot in the array.
2. During our linear search, we find the ***first value that's greater than*** the value we're asked to insert. The "correct position" is "before that greater value" but because of the way the add(int index, E element) method works on ArrayList<E>, we want to use the index of that "greater value" itself.
3. Our linear ***search finishes without finding*** a greater value. The "correct index" is "the length of the array" => append the value at the end.

OrderedArrayListSet

Lets extend ArrayListSet to an OrderedArrayListSet. find() will always return the correct index for the value, regardless of whether the value is in the set or not

What is the "correct index" for a value? Here are the possible cases:

1. The set was ***empty*** before this insertion. The "correct index" for the value is ***0*** in this case, the first spot in the array.
2. During our linear search, we find the ***first value that's greater than*** the value we're asked to insert. The "correct position" is "before that greater value" but because of the way the add(int index, E element) method works on ArrayList<E>, we want to use the index of that "greater value" itself.
3. Our linear ***search finishes without finding*** a greater value. The "correct index" is "the length of the array" => append the value at the end.

```
public class OrderedArrayListSet<T extends Comparable<T>>
                                    implements OrderedSet<T> {
private int find(T t) {
  for (int i = 0; i < this.data.size(); i++) {
    if (this.data.get(i).compareTo(t) >= 0) {
      return i;
    }
  }
  return this.data.size();
}
```

OrderedArrayListSet

```
public class OrderedArrayListSet<T extends Comparable<T>>
                                        implements OrderedSet<T> {
private int find(T t) {
  for (int i = 0; i < this.data.size(); i++) {
    if (this.data.get(i).compareTo(t) >= 0) {
      return i;
    }
  }
  return this.data.size();
}

// helper method to check if the find() value is actually the data
// or the slot it should go next. O(1) time
private boolean found(int p, T t) {
  return p < this.data.size() && this.data.get(p).equals(t);
}

// if not already there, insert at the correct sorted position
public void insert (T t ) {
  int p = this.find(t);
  if (this.found(p, t)) { return; }
  this.data.add(p, t);
}
```

Testing!

```
private void printData() {
  for (int i = 0; i < this.data.size(); i++) {
    System.out.println("data[" + i + "]: " + this.data.get(i));
  }
}

public static void main(String[] args) {
  OrderedArrayListSet<Integer> s = new OrderedArrayListSet();
  s.insert(42);
  s.insert(100);
  s.insert(3);
  s.insert(200);
  s.insert(1);
  s.printData();
}
```

```
$ java OrderedArrayListSet
data[0]: 1
data[1]: 3
data[2]: 42
data[3]: 100
data[4]: 200
```

OrderedArrayListSet

We claimed OrderedSet was better because they could go much faster, but we are still using a linear scan to find anything. How can we do better?

```
public class OrderedArrayListSet<T extends Comparable<T>>
                                    implements OrderedSet<T> {
private int find(T t) {
  for (int i = 0; i < this.data.size(); i++) {
    if (this.data.get(i).compareTo(t) >= 0) {
      return i;
    }
  }
  return this.data.size();
}
```

With a binary search, find() will complete in O(lg n) instead of O(n)

If there are 1K items to search, will only take 10 steps to find ☺
If there are 1M items to search, will only take 20 steps to find ☺ ☺
If there are 1B items to search, will only take 30 steps to find ☺ ☺ ☺

Insert() will still need to slide things over in O(n) but at least we will find the correct position in O(lg n) time

Binary Search

Binary search is conceptually easy to understand, but notoriously difficult to implement correctly. Famously first described in 1946, but not correctly published until 1961

```
private int find(T t) {

   // do me

}
```

Binary Search

Binary search is conceptually easy to understand, but notoriously difficult to implement correctly. Famously first described in 1946, but not correctly published until 1961

```
private int find(T t) {
  int l = 0, u = this.data.size()-1;

  while(l <= u) {
    int m = (l + u) / 2;

    if (this.data.get(m).compareTo(t) > 0) {
      u = m - 1;

    } else if (this.data.get(m).compareTo(t) == 0) {
      return m;

    } else {
      l = m + 1;
    }
  }
  return l;
}
```

Binary Search

Binary search is conceptually easy to understand, but notoriously difficult to implement correctly. Famously first described in 1946, but not correctly published until 1961

```
private int find(T t) {
  int l = 0, u = this.data.size()-1;

  while(l <= u) {
    int m = (l + u) / 2;

    if (this.data.get(m).compareTo(t) > 0) {
      u = m - 1;

    } else if (this.data.get(m).compareTo(t) == 0) {
      return m;

    } else {
      l = m + 1;
    }
  }
  return l;
}
```

Invariant: always searching within [l,u]

While non-empty range

Pick midpoint, integer arithmetic

m > t, check first half excluding m

Eureka!

Must be in the bottom, excluding m

Not found, l > u

Testing

```
$ seq 1 100000 | awk '{print int(rand()*100000)}' > rand100k.txt
```

```
$ time java UniqueArrayListSet < rand100k.txt > /dev/null

real    0m8.740s
user    0m9.053s
sys     0m0.369s
```

```
$ time java UniqueOrderedArrayListSetFast < rand100k.txt > /dev/null

real    0m1.199s
user    0m2.005s
sys     0m0.260s
```

Substantial speedups by replacing one function with another

Much more substantial than the move-to-front heuristics we saw

Next Steps

1. Work on HW5
2. Check on Piazza for tips & corrections!