CS 600.226: Data Structures

Michael Schatz

Oct 24, 2018

Lecture 24. BitSets

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!

HW6

Assignment 6: Setting Priorities

Out on: October 26, 2018

Due by: November 2, 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 sixth assignment is all about sets, priority queues, and various forms of experimental analysis aka benchmarking. You'll work a lot with jaybee as well as with new incarnations of the old Unique program. Think of the former as "unit benchmarking" the individual operations of a data structure, think of the latter as "system benchmarking" a complete (albeit small) application.

Agenda

1. ***Recap on Priority Queues and Heaps***

2. ***BitSets***

Part 1.1: Priority Queues

Queues

Whenever a resource is shared among multiple jobs:

- accessing the CPU
- accessing the disk
- Fair scheduling (ticketmaster, printing)

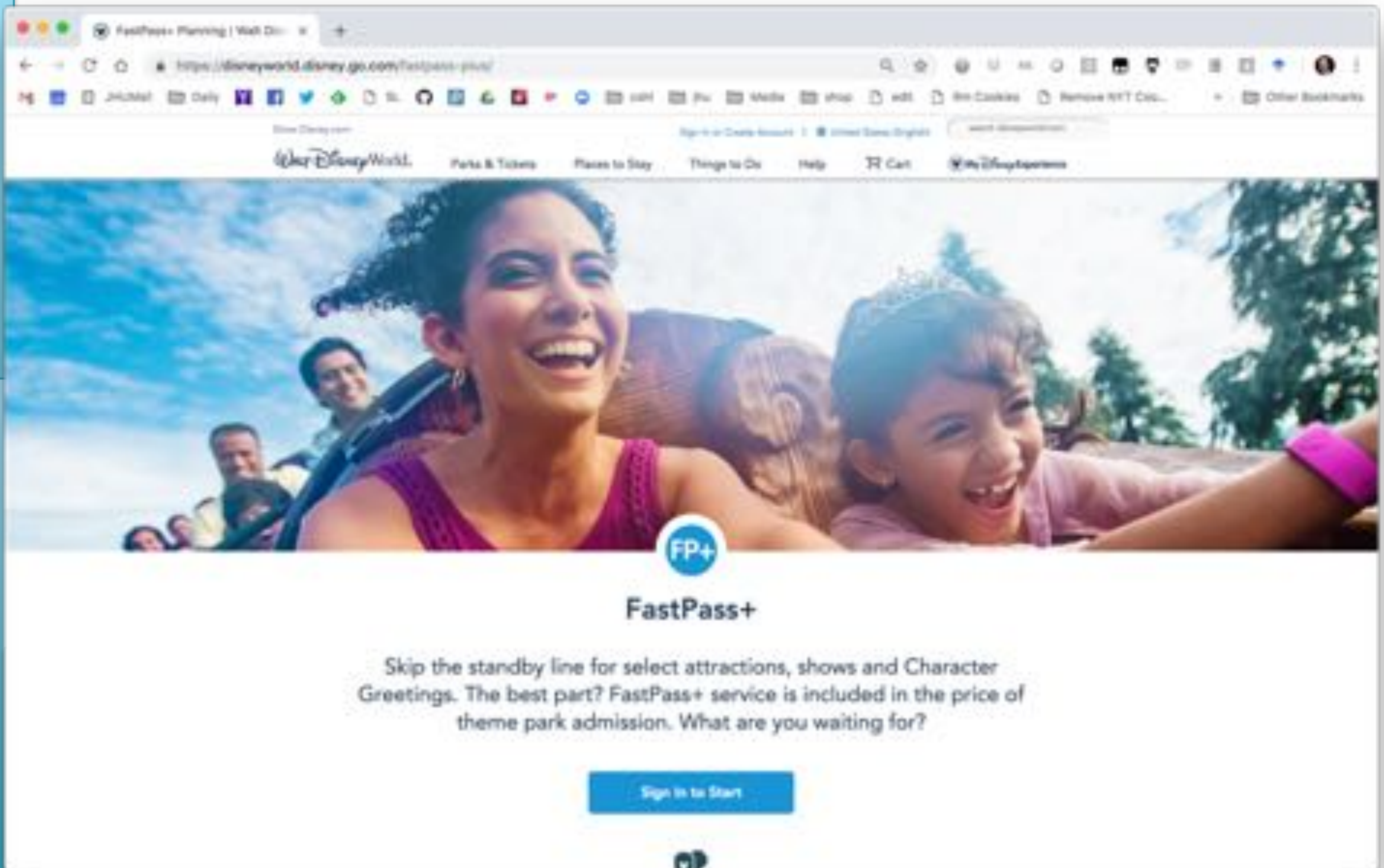
Whenever data is transferred asynchronously (data not necessarily received at same rate as it is sent):

- Sending data over the network
- Working with UNIX pipes:
  - ./slow | ./fast | ./medium

Also many applications to searching graphs (see 3-4 weeks)

FIFO: First-In-First-Out
Add to back +
Remove from front

Priority Queues



Priority Queue Interface

```
public interface PriorityQueue<T extends Comparable<T>> {
    void insert(T t);
    void remove() throws EmptyQueueException;
    T top() throwsEmptyQueueException;
    boolean empty();
}
```

Similar to a regular Queue, except the top() returns the ”largest” item rather than the first item inserted (top() instead of front())

```
pq.insert(42);
pq.insert(3);
pq.insert(100);
while (!pq.empty()){
  System.out.println(pq.top());
  pq.remove();
}
```

```
Prints:

100
42
3
```

What data structure should we use to implement a PQ?

An OrderedSet (using Binary Search :-))

Although we would allow for duplicates in a PQ

Priority Queue of Fruit

What if we wanted to use a Priority Queue of Fruit

```
PriorityQueue<Fruit> fpq = new PriorityQueue<Fruit>();
fpq.insert(apple);
fpq.insert(tomato);
fpq.insert(grape);
while (!pq.empty()){
  System.out.println(pq.top());
  pq.remove();
}
```

Prints:	**Value:**
tomato	$58B
grape	$39B
apple	$32B

How is the sort order defined?

Fruit class must implement/extend the Comparable interface by implementing the compareTo() method.

```
public class Fruit {
  int compareTo(Fruit other) {
    return this.globalValue - other.globalValue;
  }
}
```

Priority Queue Sort Order

What if we wanted to retrieve Integers sorted from smallest to largest?

1. Rewrite the priority queue: MinPriorityQueue, MaxPriorityQueue ¯_(ツ)_/¯

2. Change the comparison function ☺

Integers implement the compareTo() method:

Returns the value 0 if this Integer is equal to the argument Integer; a value less than 0 if this Integer is numerically less than the argument Integer; and a value greater than 0 if this Integer is numerically greater than the argument Integer (signed comparison).

https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html

Extend the Priority Queue interface to accept a functor (function object) to establish the sort order

```
Interface Comparator<T> {
    int compare(T o1, T o2)
    boolean equals(Object obj)
}
```

```
class SortAscending<T>
        implements Comparator<T> {
    int compare(T o1, T o2) {
        //return o1.compareTo(o2)
        return o2.compareTo(o1);
    }
}
```

```
PriorityQueue<> p = new PriorityQueue<Integer>(new SortAscending());
```

Priority Queue Implementation

```
pq.insert(42);
pq.insert(3);
pq.insert(100);
while (!pq.empty()){
  System.out.println(pq.top());
  pq.remove();
}
```

```
f[]b
f[42]b
f[42,3]b
f[100,42,3]b
f[42,3]b
f[3]b
f[]b
```

PQ implemented with an OrderedArrayListSet has some hidden costs:
Insert: O(lg n + n) time to find() then slide into correct location
Remove: O(n) time: slide items over

What can we do to improve this?

```
b[]f
b[42]f
b[3,42]f
b[3,42,100]f
b[3,42]f
b[3]f
b[]f
```

Ordering from back to front in the array allows for O(1) remove(), although insert() will remain at O(lg n + n)

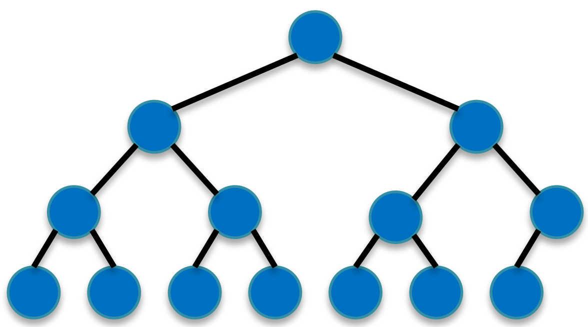
What else can we do?

Do we need all the items sorted all the time?

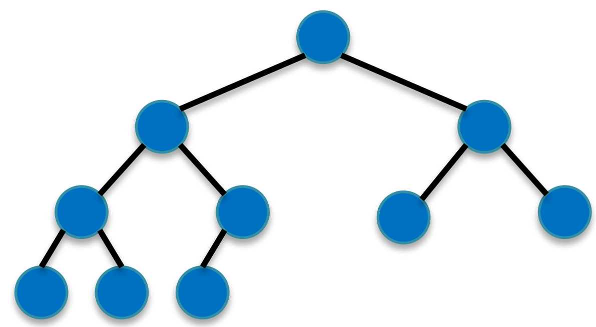
Part 1.2: (Binary) Heaps

Binary Heaps

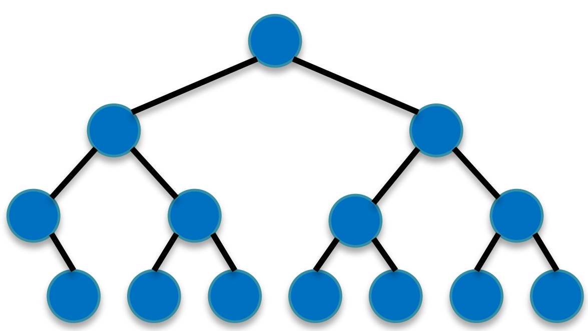
Shape Property:
Complete binary tree with every level full, except potentially the bottom level,
AND bottom level filled from left to right



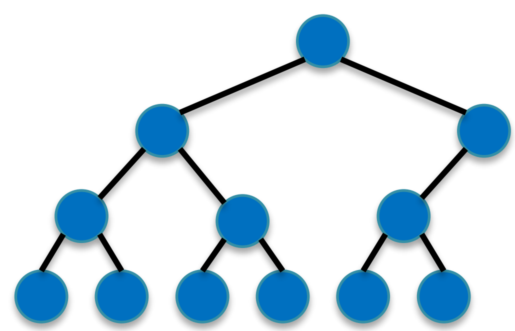
Valid



Valid



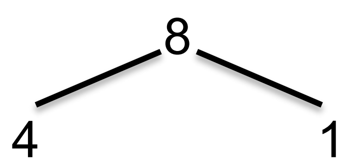
Invalid



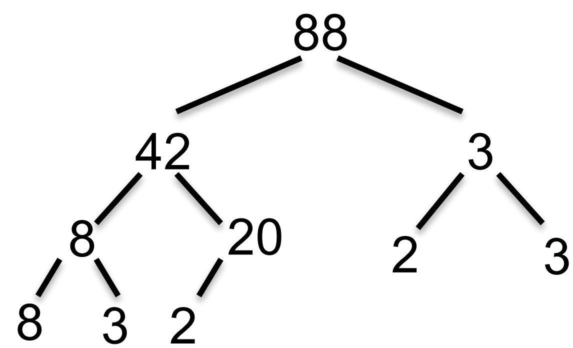
Invalid

Binary Heaps

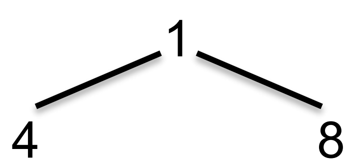
Ordering Property:
The value of each node is greater than or equal to the value of its children,
BUT there is no ordering between left and right children



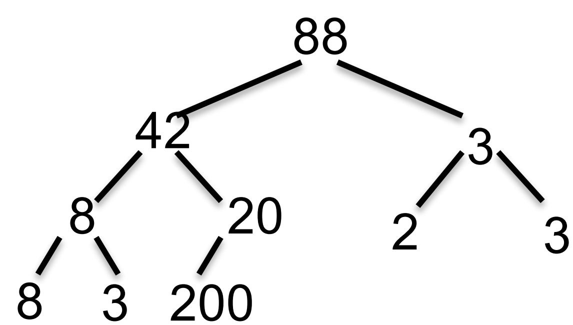
Valid



Valid

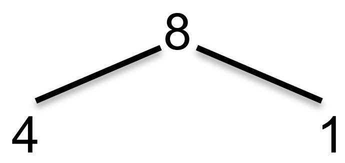


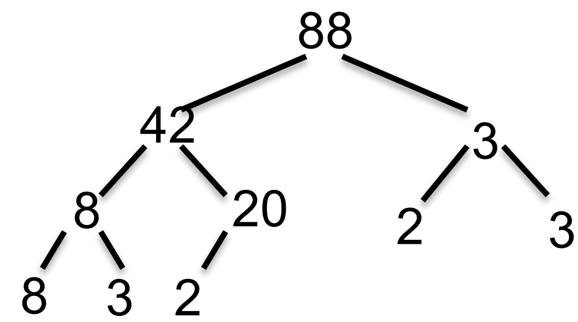
Invalid



Invalid

Binary Heaps





What does the shape property imply about the height of the tree?

Guaranteed to be lg n ☺

What does the ordering property imply about the top() of the tree?

Guaranteed max value will be in the root node

That's interesting, I wonder if we could use this for a priority queue…

… just need to efficiently insert() and removeTop()

Inserting into a binary heap

Insert the elements 8, 2, 7, 4

```
null      8        8           8            8           8
                  /           / \          / \         / \
                 2           2   7        2   7       4   7
                                         /           /
                                        4           2

        i(8)     i(2)         i(7)                 i(4)
```

The ***shape property*** tells us that we need to fill one level at a time, from left to right. So the ***number of elements*** in a heap ***uniquely determines where the next node*** has to be placed.

What about the ***ordering property***? When we insert 4, the parent 2 is not ≥ 4, so the ***ordering property is violated***. There's an ***easy fix*** however, just swap the values!

Note that in general, we ***may need to keep swapping "up the tree"*** as long as the ordering property is still violated. ***But since there are only log n levels, this can take at most O(log n) time in the worst case.***

Remove top from a binary heap

Remove the top

```
      8
     / \
    4   7
   /
  2
```

```
    4   7
   /
  2
```

ERROR:
2 trees

```
    4
   / \
  2   7
```

ERROR:
4 < 7

```
      7
     /
    4
   /
  2
```

ERROR:
Shape Violation

Any ideas?

1.Swap last

```
      2
     / \
    4   7
   /
  8
```

2. Remove last

```
    2
   / \
  4   7
```

3. Swap down from root with larger child

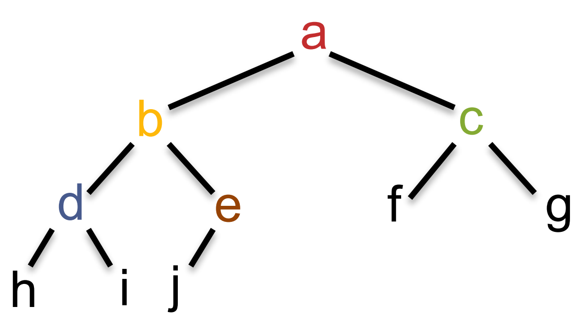
```
    7
   / \
  4   2
```

Note that in general, we ***may need to keep swapping "down the tree"*** as long as the ordering property is still violated. ***But since there are only log n levels, this can take at most O(log n) time in the worst case.***

Heap Implementation

We could implement a heap as a tree with references, but those references take up a lot of space and are relatively slow to resolve

Lets encode the tree inside an array!



1	2	3	4	5	6	7	8	9	10
a	b	c	d	e	f	g	h	i	j

Encoding a complete tree into the array in <u>level order</u> puts the children and parent in predictable locations (Math is easier if the array starts at 1 instead of 0)

Parent(i) = array[i/2]
Parent(f) = parent(6) = array[6/2] = array[3] = c

left(i) = array[i*2] & right(i) = array[i*2+1]
left(3) = array[3*2] = array[6] = f & right(3) = array[3*2+1] = array[7] = g

Heap-based Priority Queue

```
pq.insert(42);
pq.insert(3);
pq.insert(100);
while (!pq.empt
  System.out.pr
  pq.remove();
}
```

```
[]
           add 42 at end & upheap
[42]
           add 3 at end & upheap
[42,3]
           add 100 at end
[42,3,100]
           upheap 100
[100,3,42]
           remove top: swap root
[42,3,100]
           remove top: remove last & downheap
[42,3]
           remove top: swap root
[3,42]
           remove top: remove last & downheap
[3]
           remove top
[]
```

Heap-based Priority Queue

```
pq.insert(42);
pq.insert(3);
pq.insert(100);
while (!pq.empt
  System.out.pr
  pq.remove();
}
```

```
[]
              add 42 at end & upheap
[42]
              add 3 at end & upheap
[42,3]
              add 100 at end
[42,3,100]
              upheap 100
[100,3,42]
              remove top: swap root
[42,3,100]
                                                 heap
              remove top: swap root
              remove top: remove last & downheap
```

Seems a little complicated, but each insert completes in O(lg n) and each remove completes in O(lg n) ☺

How could you use this for a general sort routine?

Add all elements in O(n lg n); remove in sorted order in O(n lg n)

Total time for HeapSort: O(n lg n) ☺

UniqueQueue

```
import java.util.Scanner;

public final class UniqueQueue {
    private static PriorityQueue<Integer> data;
    private UniqueQueue() { }

    public static void main(String[] args) {
        data = new BinaryHeapPriorityQueue<Integer>();
        Scanner scanner = new Scanner(System.in);

        while (scanner.hasNextInt()) {
            int i = scanner.nextInt();
            data.insert(i);
        }

        Integer last = null;

        while (!data.empty()) {
            Integer i = data.remove();
            if (last == null || i != last) {
                System.out.println(i);
            }
            last = i;
        }
    }
}
```

Since data are in sorted ordered, just check to see current if different from last item

Testing

$ seq 1 1000000 | awk '{print int(rand()*1000000)}' > rand1000k.txt

```
$ time java UniqueOrderedArrayListSetFast < rand1000k.txt > /dev/null

real    0m18.386s
user    0m19.258s
sys      0m0.698s
```

```
$ time java UniqueQueue < rand1000k.txt > /dev/null

real    0m5.785s
user    0m6.912s
sys      0m1.023s
```

Substantial speedups replacing OrderedSet (with binary search but slow insert) with Heap-based Priority Queue (with O(n lg n) overall time) ☺ ☺ ☺

Part 2. IntegerSets and BitSets

Set of Integers

UnorderedSet:
- has: O(n)
- insert: O(n)
- remove: O(n)

OrderedSet (Binary Search)
- has: O(lg n)
- insert: O(lg n + n)
- remove: O(lg n + n)

PriorityQueue (Heap)
- has: O(n)
- insert: O(lg n)
- removeTop: O(lg n)

Could we do better for integers?

IntegerSet

```
Set iset = new IntegerSet();
iset.insert(3);
iset.insert(6);
iset.insert(2);
iset.insert(3)

iset.has(8);
iset.remove(2);

for(Integer i: iset) {
  System.out.println(i);
}
```

Lets assume values are between 0 and 9

Array of Boolean could work in O(1) but:

How many Booleans?

Wont this require a lot of space?

new()

0	1	2	3	4	5	6	7	8	9
F	F	F	F	F	F	F	F	F	F

insert(3)

0	1	2	3	4	5	6	7	8	9
F	F	F	**T**	F	F	F	F	F	F

insert(6)

0	1	2	3	4	5	6	7	8	9
F	F	F	**T**	F	F	**T**	F	F	F

insert(2)

0	1	2	3	4	5	6	7	8	9
F	F	**T**	**T**	F	F	**T**	F	F	F

insert(3)

0	1	2	3	4	5	6	7	8	9
F	F	**T**	**T**	F	F	**T**	F	F	F

remove(2)

0	1	2	3	4	5	6	7	8	9
F	F	**F**	**T**	F	F	**T**	F	F	F

How many Booleans?

We could perhaps use the array doubling technique, but then we wont have guaranteed O(1) insert, and will sometimes have O(n) insert time ☹

What can we do instead?

Preallocate an array covering the range of data :-)

```
// data are between 0 and 9
Set iset = new IntegerSet(10);
```

0	1	2	3	4	5	6	7	8	9
F	F	F	F	F	F	F	F	F	F

What if the range includes negative numbers?

```
this.data[-42] = True; //Error
```

`this.lo = -3`

-3	-2	-1	0	1	2	3	4	5	6
0	1	2	3	4	5	6	7	8	9
F	F	F	F	F	F	F	F	F	F

Preallocate an array over the active range:

```
Set iset = new IntegerSet(-3,6);
```

```
iset.insert(-2);
=> this.data[idx – this.lo] = True;
```

0	1	2	3	4	5	6	7	8	9
F	T	F	F	F	F	F	F	F	F

Wont this take a lot of space?

If the set is sparsely filled, most cells will be False

What can we do instead?

Using a SparseArray will save space but then we wont guarantee O(1) time

How much space will we need to store:

0 through 999,999	1M Booleans	=> 1M bytes
-500,000 through +499,999:	1M Booleans	=> 1M bytes
0 through 999,999,999	1B Booleans	=> 1G bytes

1GB per Billion values isn’t too bad, how will this look?

SimpleIntegerSet (1)

```
import java.lang.Iterable;
import java.util.Iterator;

public class SimpleIntegerSet implements IntegerSet {
    private boolean[] data;
    private int low;
    private int high;

    public SimpleIntegerSet(int low, int high) {
        if (low > high){
            throw new IllegalArgumentException("low " +
                                 low + " must be <= high " + high);
        }

        this.data = new boolean[high - low + 1];
        this .low = low;
        this.high = high;
    }

    public SimpleIntegerSet(int size) {
       this (0, size - 1);
    }
…
```

Java array of boolean primitive type

Helper constructor for positive numbers only

SimpleIntegerSet (2)

```
…
    private int index(int i) {
        if (this.low<=i&&i<=this.high) {
            return i + this.low;
        } else {
             throw new IndexOutOfBoundsException("element " +
                           i + " must be >= low " + this.low +
                           " and <= high " + this.high);
        }
    }

    private void put(int i, boolean b) {
        this.data[this.index(i)] = b;
    }

    private boolean get(int i) {
        return this.data[this.index(i)];
    }
…
```

Private methods that directly update this.data

SimpleIntegerSet (3)

```
…
    public void insert(int i) { this.put(i, true); }
    public void remove(int i) { this.put(i, false); }
    public boolean has(int i) { return this.get(i); }
    public int low()  { return this.low; }
    public int high() { return this.high; }

    // homework :-)
    public Iterator<Integer> iterator() { return null; }
}
```

Public methods

This is works in O(1), but is there anything we can do to reduce memory requirements?

Wastes a lot of space to use entire bytes for booleans!

Generally inefficient to access individual bits of memory, no bit datatype in Java

Binary Arithmetic

Integers (and all data) are really stored as a sequence of 0s and 1s

```
public class PrintBits {
  public static void main(String[] args) {
    Integer i = Integer.parseInt(args[0]);
    System.out.println("Integer: " + i +
                       " Bits: " + Integer.toBinaryString(i));
  }
}
```

```
$ java PrintBits 0
Integer: 0 Bits: 0
$ java PrintBits 1
Integer: 1 Bits: 1
$ java PrintBits 2
Integer: 2 Bits: 10
$ java PrintBits 3
Integer: 3 Bits: 11
$ java PrintBits 4
Integer: 4 Bits: 100
$ java PrintBits 8
Integer: 8 Bits: 1000
```

Binary Arithmetic

Integers (and all data) are really stored as a sequence of 0s and 1s

```
public class PrintBits {
  public static void main(String[] args) {
    Integer i = Integer.parseInt(args[0]);
    System.out.println("Integer: " + i +
                       " Bits: " + Integer.toBinaryString(i));
  }
}
```

```
$ java P
Integer:
$ java P
Integer:
$ java P
Integer:
$ java P
Integer:
$ java P
Integer:
$ java P
Integer:
```

```
$ java PrintBits 1024
Integer: 1024 Bits: 10000000000
$ java PrintBits 424242
Integer: 424242 Bits: 1100111100100110010
$ java PrintBits 1000000
Integer: 1000000 Bits: 11110100001001000000
$ java PrintBits 1048576
Integer: 1048576 Bits: 100000000000000000000
$ java PrintBits 42424242
Integer: 42424242 Bits: 10100001110101011110110010
$ java PrintBits 1073741824
Integer: 1073741824 Bits: 1000000000000000000000000000000
```

Binary Arithmetic

Java Integers are 32 bit values

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	0	0	1	1	1	0	1	0	1	0	1	1	1	1	1	0	1	1	0	0	1	0

Bits are numbered from rightmost (0) to leftmost (31)

Aka Most significant bit first (leftmost bit determines billions)

Binary:

$$
\begin{aligned}
101010_2 &= 1\text{x}2^5 + 0\text{x}2^4 + 1\text{x}2^3 + 0\text{x}2^2 + 1\text{x}2^1 + 0\text{x}2^0 \\
&= 1\text{x}32 + 0\text{x}16 + 1\text{x}8 + 0\text{x}4 + 1\text{x}2 + 0\text{x}1 \\
&= 32_{10} + 0_{10} + 8_{10} + 0_{10} + 2_{10} + 0_{10} \\
&= 42_{10}
\end{aligned}
$$

Decimal:

$$
\begin{aligned}
4711_{10} &= 4\text{x}10^3 + 7\text{x}10^2 + 1\text{x}10^1 + 1\text{x}10^0 \\
&= 4\text{x}1000 + 7\text{x}100 + 1\text{x}10 + 1\text{x}1 \\
&= 4000 + 700 + 10 + 1
\end{aligned}
$$

Positive and Negative Numbers

Non-negative numbers are represented by the leftmost bit == 0

$$
\begin{aligned}
0_{10} &= 0 &&= 00000000000000000000000000000000_2 \\
1_{10} &= 2^0 &&= 00000000000000000000000000000001_2 \\
2_{10} &= 2^1 &&= 00000000000000000000000000000010_2 \\
3_{10} &= 2^1+1 &&= 00000000000000000000000000000011_2 \\
4_{10} &= 2^2 &&= 00000000000000000000000000000100_2 \\
5_{10} &= 2^2+1 &&= 00000000000000000000000000000101_2 \\
& && \;\;\ldots \\
2{,}147{,}483{,}647_{10} &= 2^{31}-1 &&= 01111111111111111111111111111111_2
\end{aligned}
$$

Negative numbers are represented by the leftmost bit == 1 using "two's complement"

$$
\begin{aligned}
-1_{10} &= -2^0 &&= 11111111111111111111111111111111_2 \\
-2_{10} &= -2^1 &&= 11111111111111111111111111111110_2 \\
-3_{10} &= -2^1+1 &&= 11111111111111111111111111111101_2 \\
-4_{10} &= -2^2 &&= 11111111111111111111111111111100_2 \\
-5_{10} &= -2^2+1 &&= 11111111111111111111111111111011_2 \\
& && \;\;\ldots \\
-2{,}147{,}483{,}648_{10} &= -2^{31} &&= 10000000000000000000000000000000_2
\end{aligned}
$$

(Bits for -x) = ~(Bits for +x) + 1

Binary Arithmetic

Two's complement allows for binary arithmetic without any additional rules

```
            Addition

 11111 111    (carry)
  0000 1111   (15)
+ 1111 1011   (-5)
===========
  0000 1010   (10)
```

```
          Subtraction

 11110 000    (borrow)
  0000 1111   (15)
- 1111 1011   (-5)
===========
  0001 0100   (20)
```

```
      Multiplication

    00000110   (6)
*   11111011   (-5)
============
         110
        1100
       00000
      110000
     1100000
    11000000
   x10000000
+ xx00000000
============
  xx11100010 (-30)
[x can be truncated]
```

https://introcs.cs.princeton.edu/java/61data/

Binary Logic

There are several common logical operations that can be applied to bits

AND (&)

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

OR (|)

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

XOR (^)

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

NOT (~)

A	~A
0	1
1	0

The operators can also be applied to several bits at once (32 for int, 64 for long)

AND (&)

```
  a = 010110
  b = 001110
a&b = 000110
```

OR (|)

```
  a = 010110
  b = 001110
a|b = 011110
```

XOR (^)

```
  a = 010110
  b = 001110
a^b = 011000
```

NOT (~)

```
 a = 010110
~a = 101001
```

Bit Shifting

In addition to logical operations, we can shift bits left (<<) or right (>>)

```
Binary                                  Decimal

000001 << 1 == 000010                   1<<1  == 2
000001 << 2 == 000100                   1<<2  == 4
000001 << 3 == 001000                   1<<3  == 8

001101 >> 1 == 000110                   13>>1 == 6
001101 >> 2 == 000011                   13>>2 == 3
001101 >> 3 == 000001                   13>>3 == 1
```

```java
public class BitShifts {
  public static void main(String[] args) {
    System.out.println("1 << 1: " + (1 << 1));
    System.out.println("1 << 2: " + (1 << 2));
    System.out.println("1 << 3: " + (1 << 3));

    System.out.println("13 >> 1: " + (13 >> 1));
    System.out.println("13 >> 2: " + (13 >> 2));
    System.out.println("13 >> 3: " + (13 >> 3));
  }
}
```

Bit Twiddling

Using these operations, we can do some pretty interesting computes

```
public class BitTwiddleEO {
  public static void main(String[] args) {
    int x = (int) Integer.parseInt(args[0]);
    int r = (x & 1);
    Boolean b = (r == 1);
    System.out.println("x: " +  x + " r: " + r + " b: " + b);
  }
}
```

```
$ java BitTwiddleEO 0
x: 0 r: 0 b: false
$ java BitTwiddleEO 1
x: 1 r: 1 b: true
$ java BitTwiddleEO 2
x: 2 r: 0 b: false
$ java BitTwiddleEO 3
x: 3 r: 1 b: true
```

```
$ java BitTwiddleEO 42
x: 42 r: 0 b: false
$ java BitTwiddleEO 99
x: 99 r: 1 b: true
$ java BitTwiddleEO -99
x: -99 r: 1 b: true
$ java BitTwiddleEO -98
x: -98 r: 0 b: false
```

X is even => false; X is odd => true

Bit Twiddling

Using these operations, we can do some pretty interesting computes

```
public class BitTwiddleA {
  public static void main(String[] args) {
    int x = (int) Integer.parseInt(args[0]);
    int y = x >> 31;
    int z = (x + y) ^ y;
    System.out.println("x: " +  x + " z: " + z);
  }
}
```

```
$ java BitTwiddleA 1
x: 1 z: 1
$ java BitTwiddleA 2
x: 2 z: 2
$ java BitTwiddleA 3
x: 3 z: 3
$ java BitTwiddleA 424242
x: 424242 z: 424242
```

```
$ java BitTwiddleA -1
x: -1 z: 1
$ java BitTwiddleA -2
x: -2 z: 2
$ java BitTwiddleA -3
x: -3 z: 3
$ java BitTwiddleA -424242
x: -424242 z: 424242
```

z=abs(x)

Bit Twiddling

Using these operations, we can do some pretty interesting computes

```
public class BitTwiddleX {
  public static void main(String[] args) {
    int x = (int) Integer.parseInt(args[0]);
    int y = (int) Integer.parseInt(args[1]);
    System.out.println("x: " +  x + " y: " + y);
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;
    System.out.println("x: " +  x + " y: " + y);
  }
}
```

```
$ java BitTwiddleX 1 2
x: 1 y: 2
x: 2 y: 1
```

```
$ java BitTwiddleX 1234 -5678
x: 1234 y: -5678
x: -5678 y: 1234
```

Swap x and y without any temporary variables

Bit Twiddling

Using these operations, we can do some pretty interesting computes

```
public class BitTwiddleC {
  public static void main(String[] args) {
    int x = Integer.parseInt(args[0]);
    System.out.println("x: " +  x);
    int c = 0;
    while (x != 0) {
       c++;
       x = x & (x - 1);
    }
    System.out.println("x: " +  x + " c: " + c);
  }
}
```

```
$ java BitTwiddleC 0
x: 0
x: 0 c: 0
$ java BitTwiddleC 1
x: 1
x: 0 c: 1
$ java BitTwiddleC 2
x: 2
x: 0 c: 1
```

```
$ java BitTwiddleC 3
x: 3
x: 0 c: 2
$ java BitTwiddleC 4
x: 4
x: 0 c: 1
$ java BitTwiddleC 4242
x: 4242
x: 0 c: 4
```

Bit Twiddling

Using these operations, we can do some pretty interesting computes

```
public class BitTwiddleC {
  public static void main(String[] args) {
    int x = Integer.parseInt(args[0]);
    System.out.println("x: " +  x);
    int c = 0;
    while (x != 0) {
       c++;
       x = x & (x - 1);
    }
    System.out.println("x: " +  x + " c: " + c);
  }
}
```

```
$ java BitTwiddleC 0 (000)
x: 0
x: 0 c: 0
$ java BitTwiddleC 1 (001)
x: 1
x: 0 c: 1
$ java BitTwiddleC 2 (010)
x: 2
x: 0 c: 1
```

```
$ java BitTwiddleC 3 (011)
x: 3
x: 0 c: 2
$ java BitTwiddleC 4 (100)
x: 4
x: 0 c: 1
$ java BitTwiddleC 4242
            (1000010010010)
x: 4242
x: 0 c: 4
```

Bit Twiddling

Using these operations, we can do some pretty interesting computes

```
public class BitTwiddleC {
  public static void main(String[] args) {
    int x = Integer.parseInt(args[0]);
    System.out.println("x: " +  x);
    int c = 0;
    while (x != 0) {
       c++;
       x = x & (x - 1);
    }
    System.out.println("x: " +  x + " c: " + c);
  }
}
```

Count how many bits are set to 1 (popcount)

```
$ java BitTwiddleC 0 (000)
x: 0
x: 0 c: 0
$ java BitTwiddleC 1 (001)
x: 1
x: 0 c: 1
$ java BitTwiddleC 2 (010)
x: 2
x: 0 c: 1
```

```
$ java BitTwiddleC 3 (011)
x: 3
x: 0 c: 2
$ java BitTwiddleC 4 (100)
x: 4
x: 0 c: 1
$ java BitTwiddleC 4242
            (1000010010010)
x: 4242
x: 0 c: 4
```

TinyIntegerSet

```
1. int iset = 0
0000000000
```

```
2. iset.insert(3);
iset = iset | (1<<3);
   0000000000
 | 0000001000
 ============
   0000001000
```

```
3. iset.insert(6);
iset = iset | (1<<6);
   0000001000
 | 0001000000
 ============
   0001001000
```

```
4. iset.insert(2);
iset = iset | (1<<2);
   0001001000
 | 0000000100
 ============
   0001001100
```

```
5. iset.insert(3);
iset = iset | (1<<3);
   0001001100
 | 0000001000
 ============
   0001001100
```

```
6. iset.has(8);
(iset & (1<<8)) != 0
   0001001100
 & 0100000000
 ============
   0000000000 => F
```

```
7. iset.has(3);
(iset & (1<<3)) != 0
   0001001100
 & 0000001000
 ============
   0000001000 => T
```

```
8. iset.remove(2);
iset = iset & ~(1<<2)

1 << 2   = 0000000100
~(1<<2)  = 1111111011

   0001001100
 & 1111111011
=============
   0001001000
```

```
9. iset.clear()
iset = 0
```

Woohoo! O(1) for everything

But only for 32 values

BitSet

Access the individual bits within an array of ints to represent an IntegerSet
All operations in O(1) like a boolean array, although uses 8x less memory ☺

```
                 0         1         2         3         4        <--- array slots
              +--------+--------+--------+--------+--------+----
int[] data:   |        |        |        |        |        | ...
              +--------+--------+--------+--------+--------+----
                0..31   32..63   64..95   96..127  128...       <--- bit positions
```

How do you figure out which array slot and bit position to access?

```
slot = k / 32
 bit = k % 32
```

```
k=27 Slot: 0 Bit: 27
```

```
k=37 Slot: 1 Bit: 37-32=5
```

```
k=61 Slot: 1 Bit: 61-32=29
```

BitSet implementation:

```
insert(x):  data[x/32] |  (1<<(x%32))
remove(x):  data[x/32] & ~(1<<(x%32))
   has(x): (data[x/32] &  (1<<(x%32))) != 0
```

How could you extend BitSet/IntegerSet to sort integers?

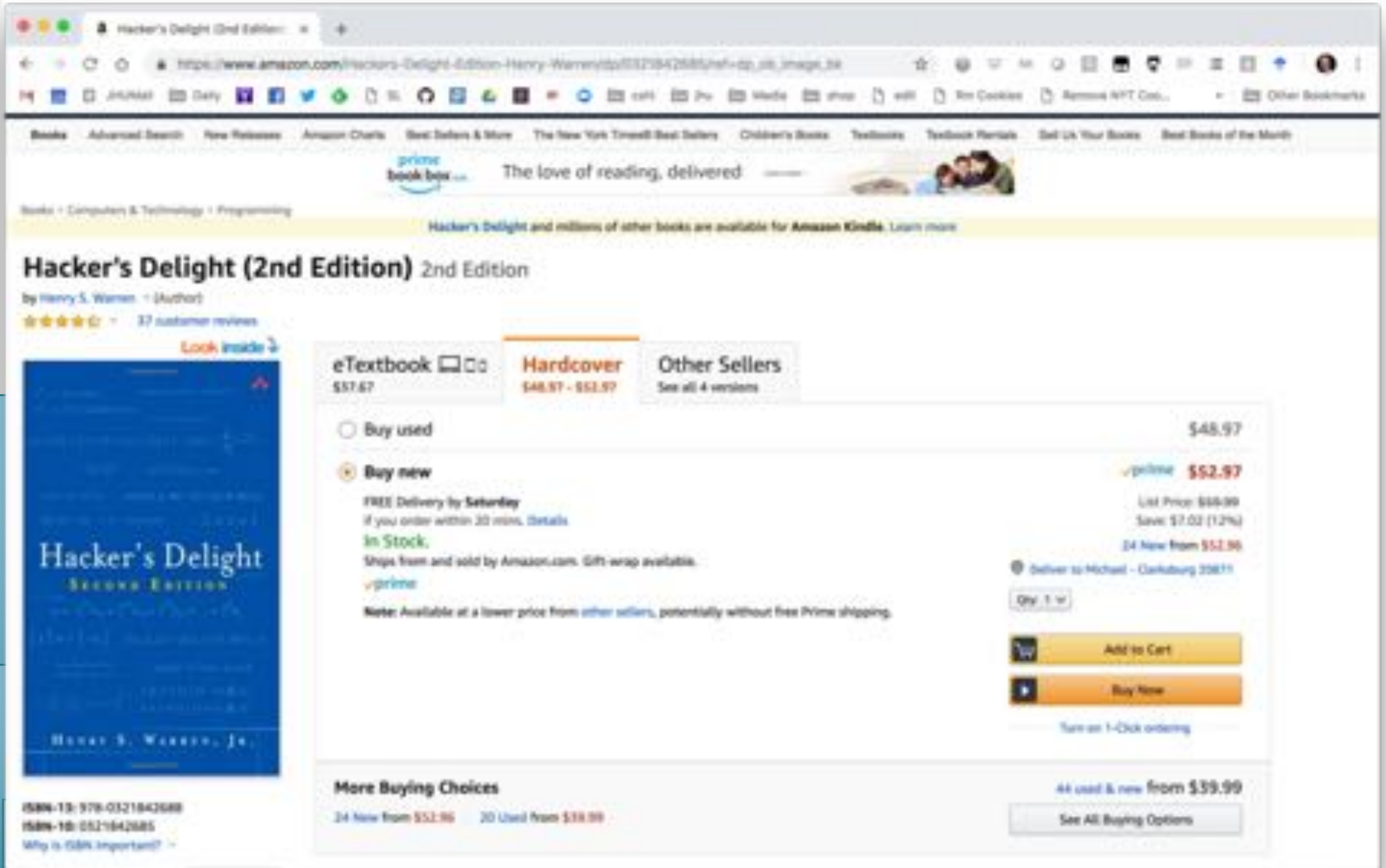
Big array of ints that count occurrences of each int; scan data to increment; scan table to output. O(n) sorting ☺

Bit Twiddling Hacks

https://graphics.stanford.edu/~seander/bithacks.html

Contents

Hacker’s Delight



Next Steps

1. Work on HW6
2. Check on Piazza for tips & corrections!