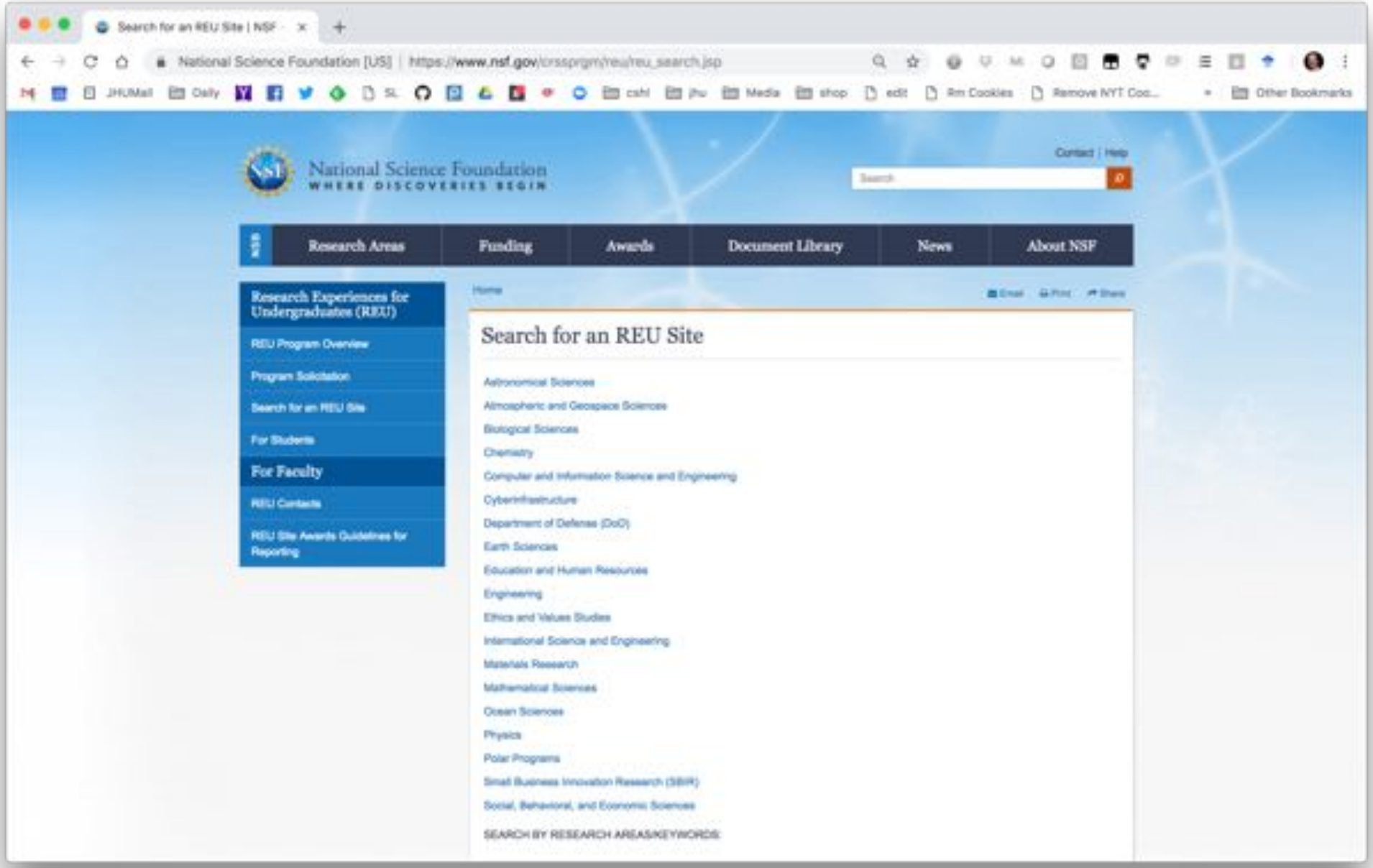
CS 600.226: Data Structures

Michael Schatz

Oct 29, 2018

Lecture 25. Maps and BSTs



https://www.nsf.gov/crssprgm/reu/reu_search.jsp

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:

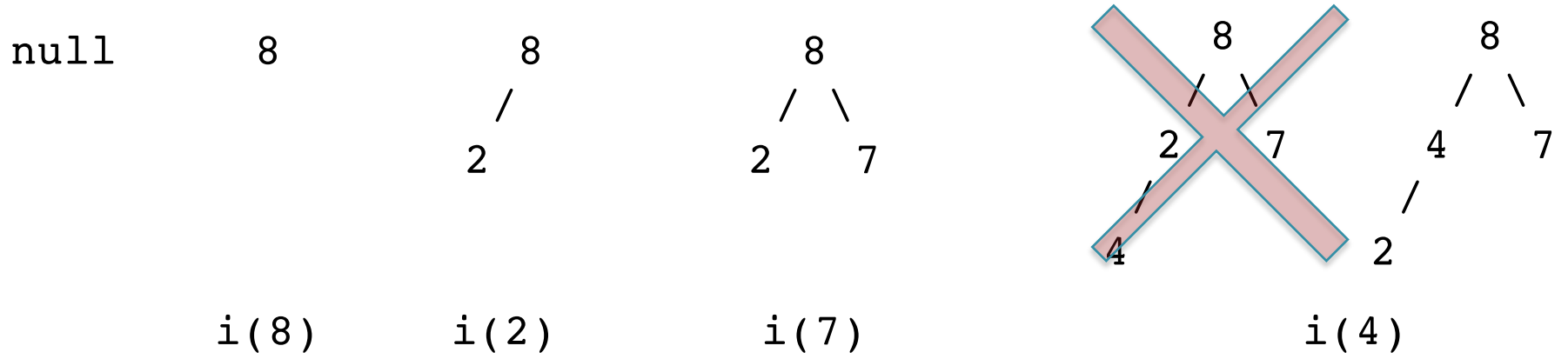
- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

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.

Inserting into a binary heap

Insert the elements 8, 2, 7, 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.

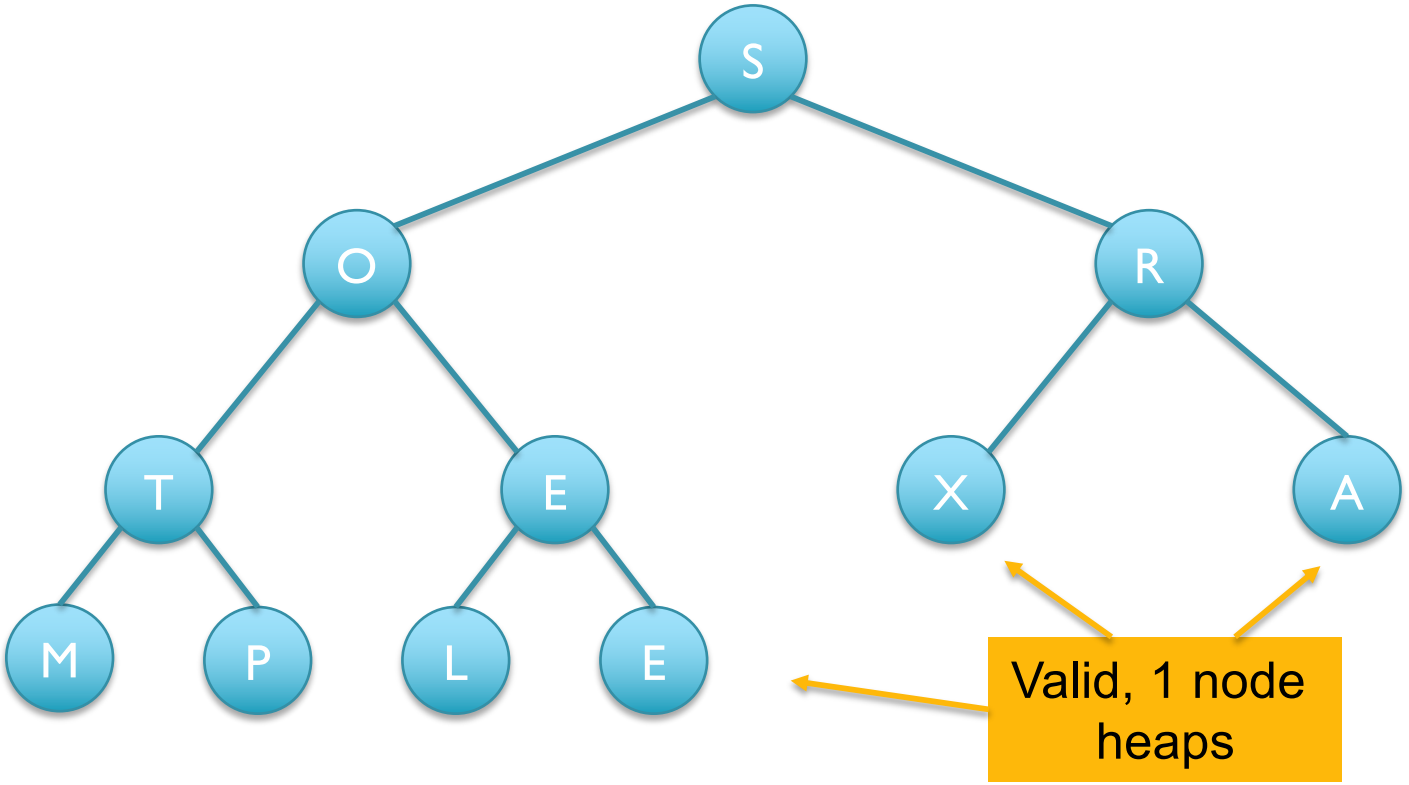
What about the ***ordering property***? When we insert 4, the parent 2 is not ≥ 4, so the ***ordering property is violated***. There's an ***easy fix*** however, just swap the values!

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.***

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

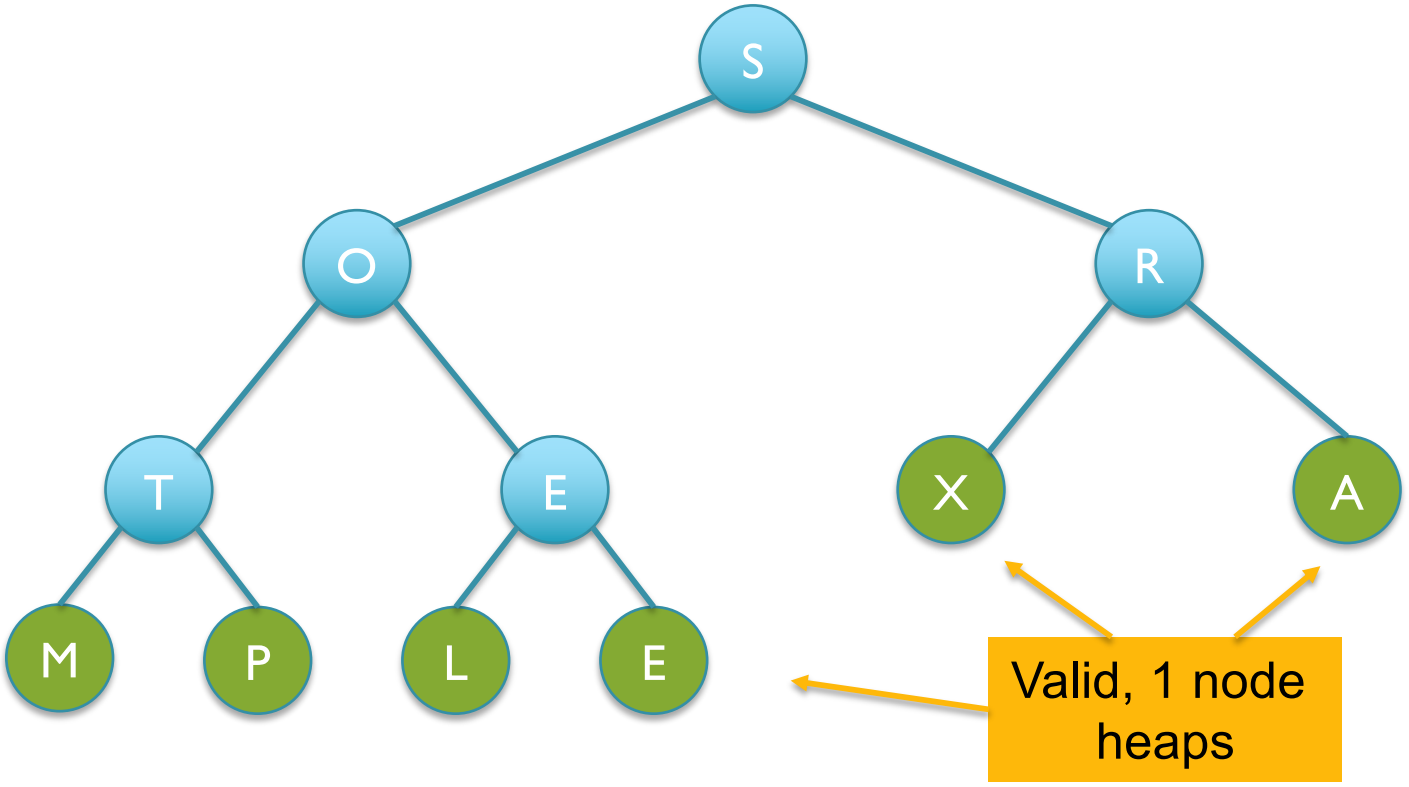
S	O	R	T	E	X	A	M	P	L	E



Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

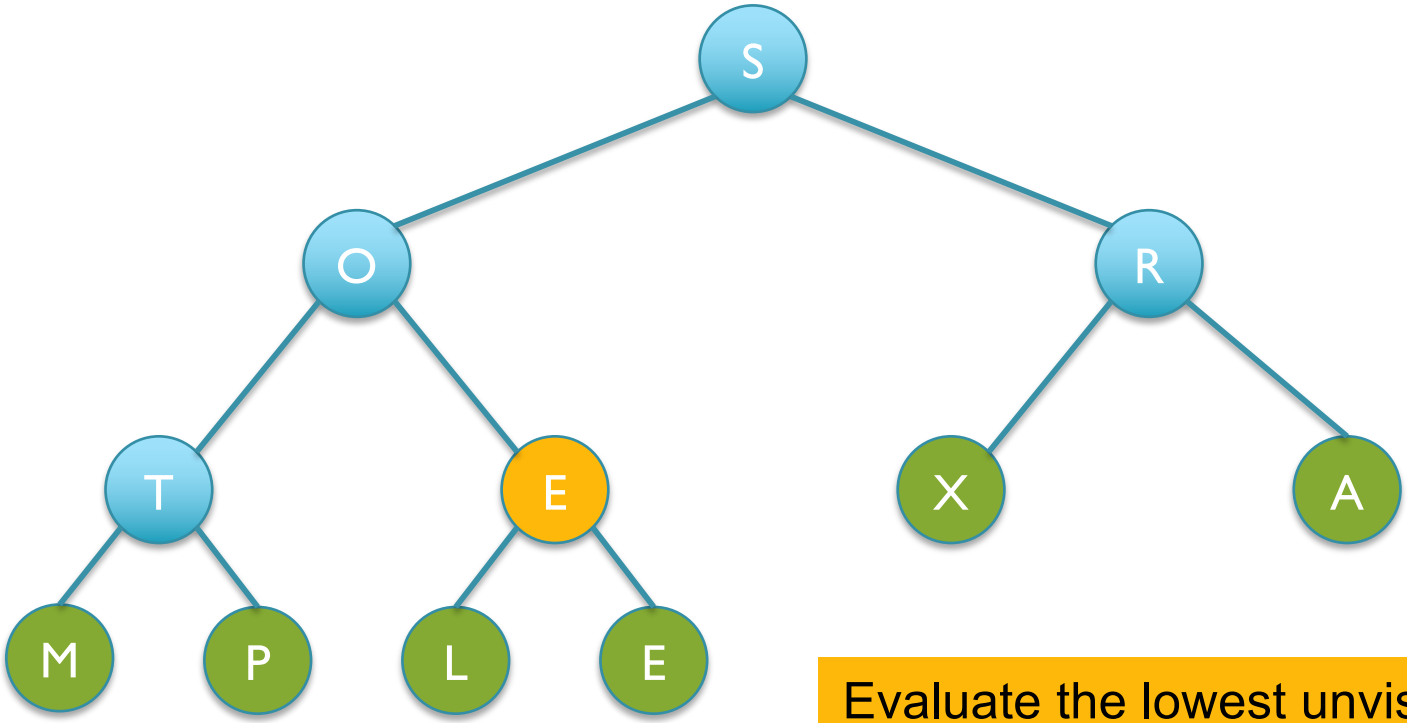
S	O	R	T	E	X	A	M	P	L	E



Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	O	R	T	E	X	A	M	P	L	E

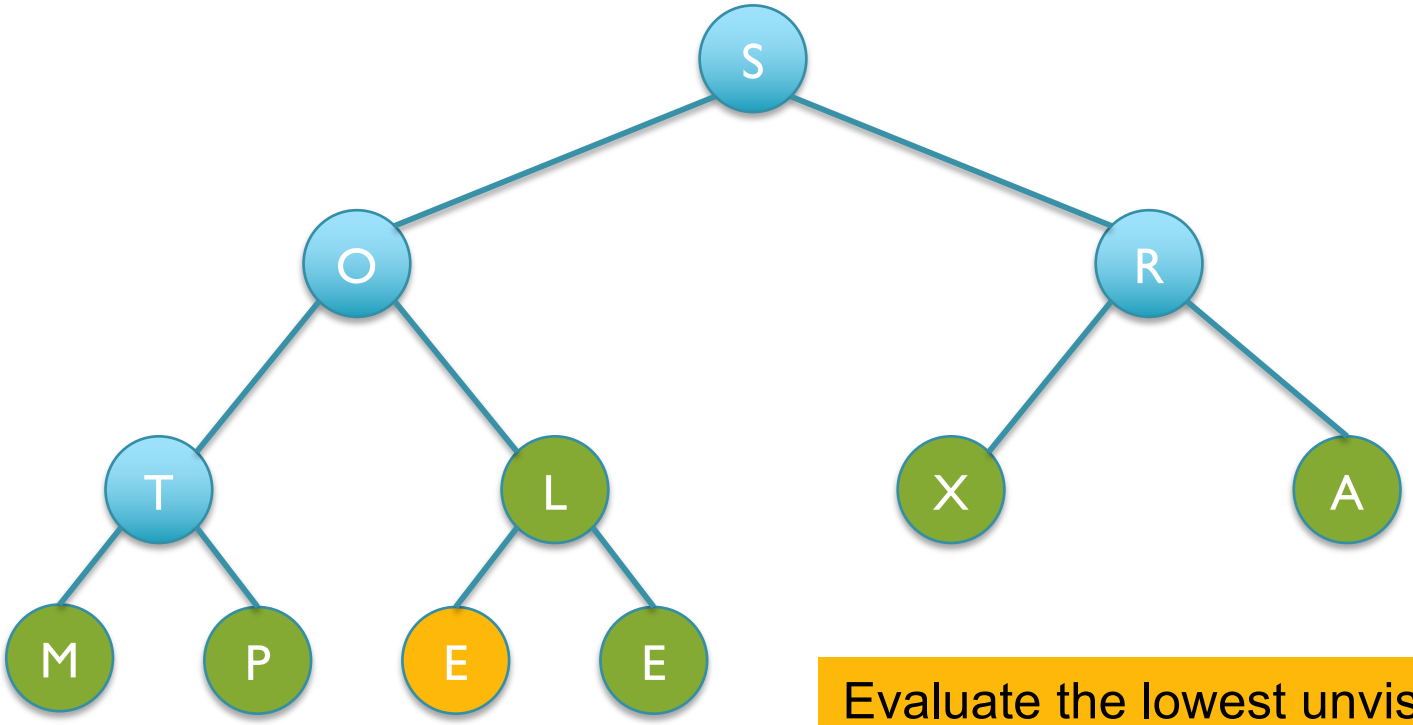


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	O	R	T	L	X	A	M	P	E	E

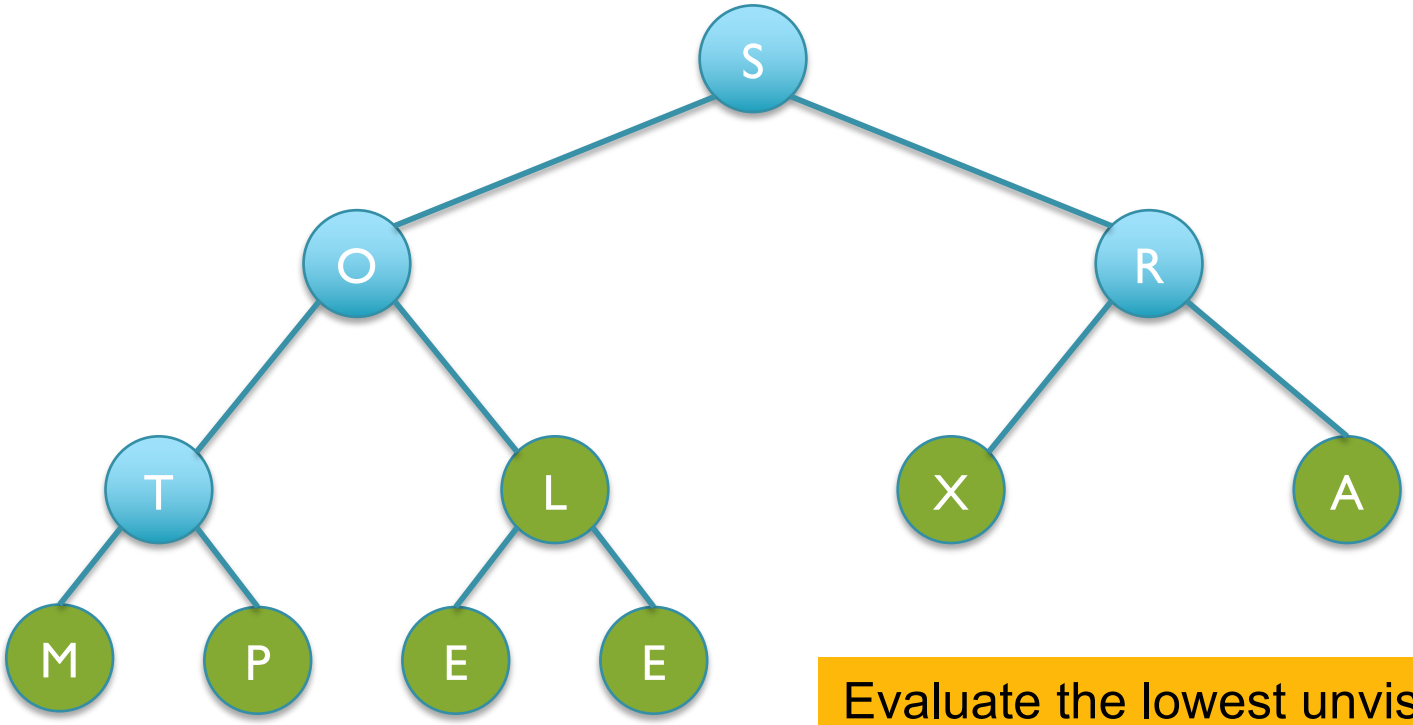


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	O	R	T	L	X	A	M	P	E	E

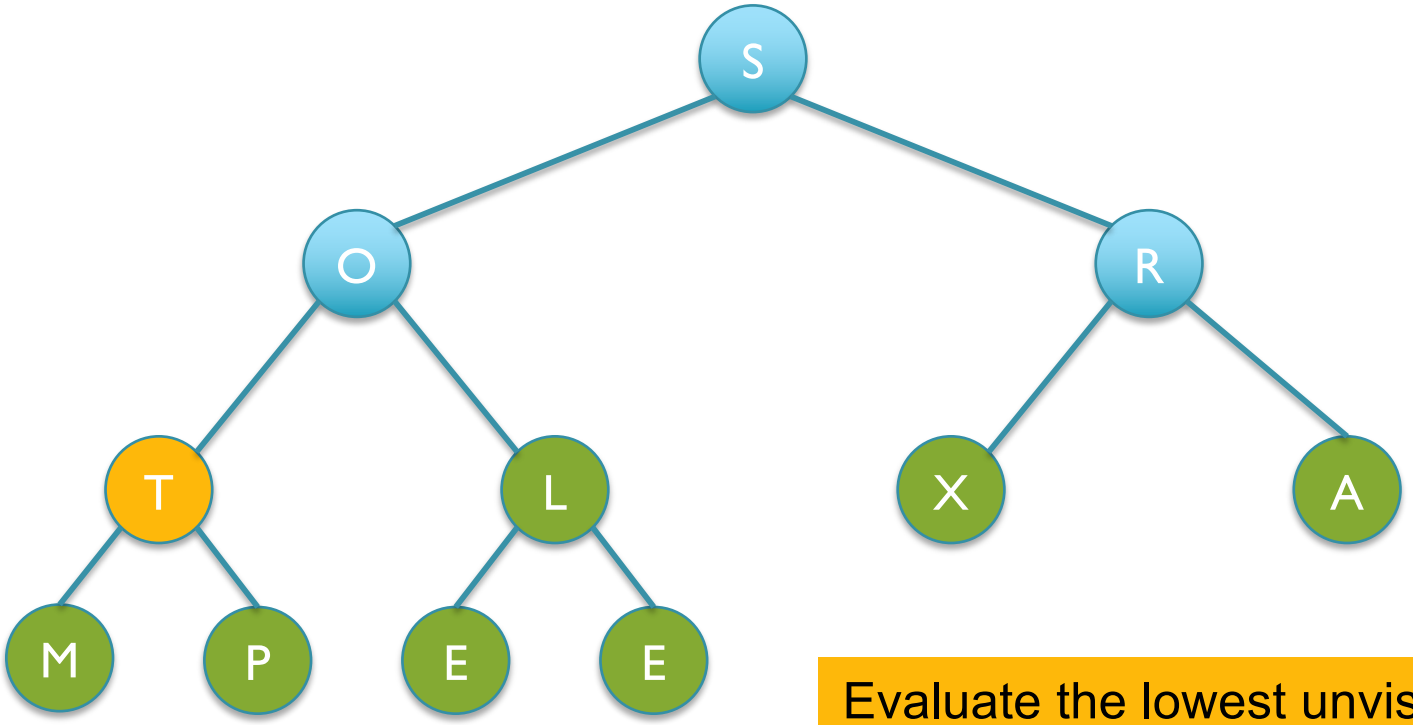


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	O	R	T	L	X	A	M	P	E	E

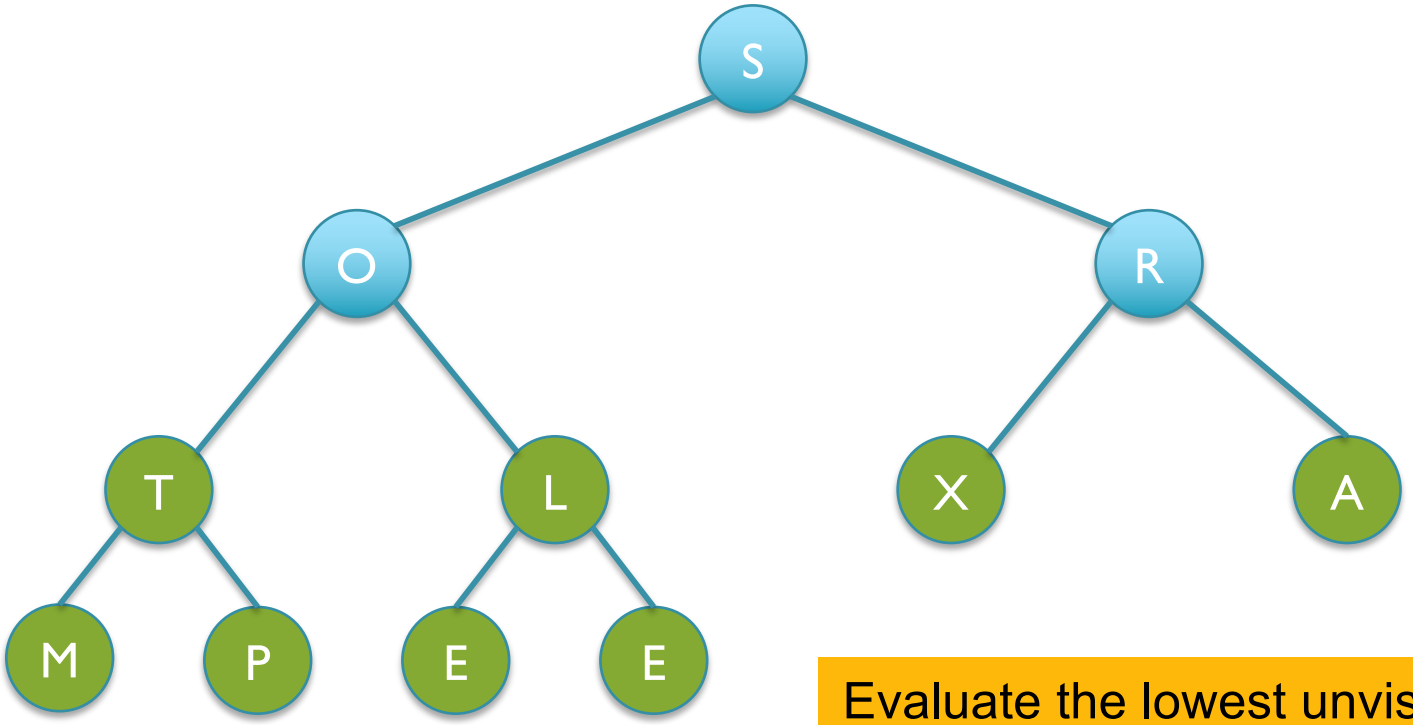


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	O	R	T	L	X	A	M	P	E	E

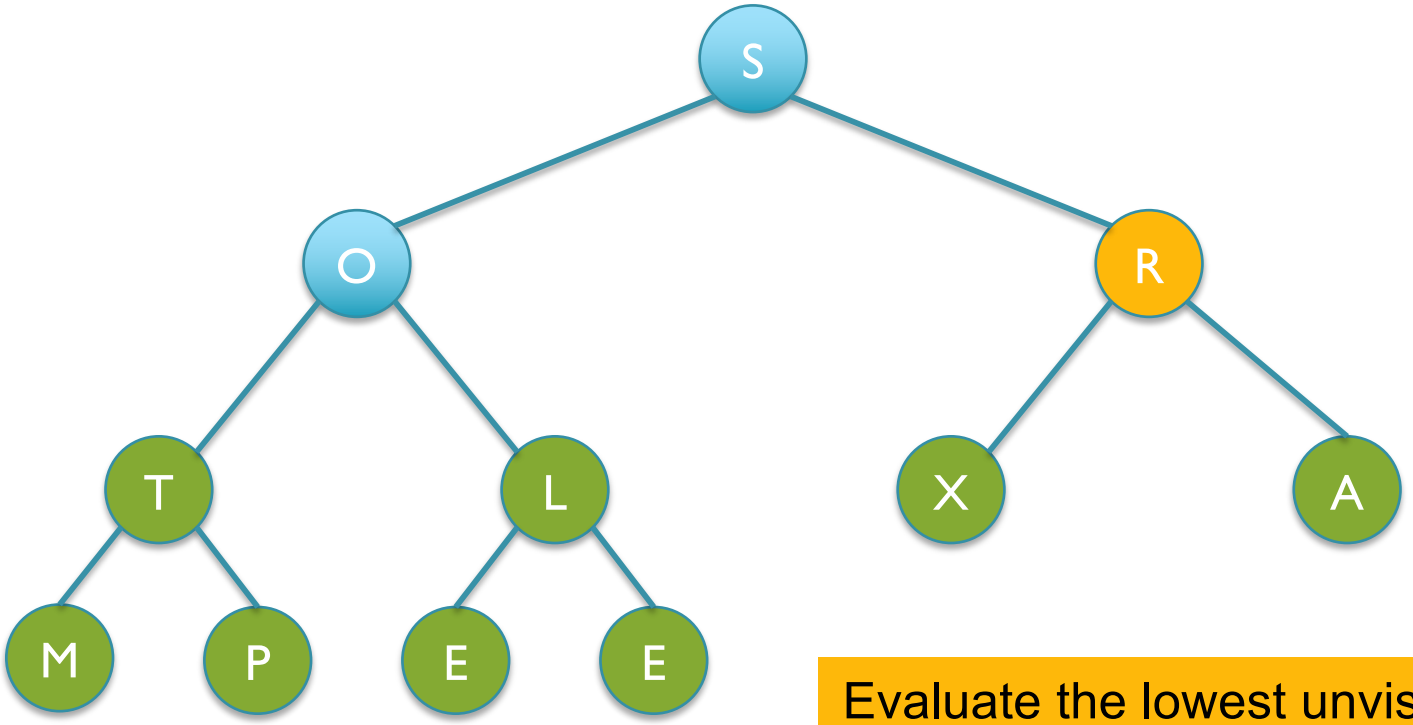


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	O	R	T	L	X	A	M	P	E	E

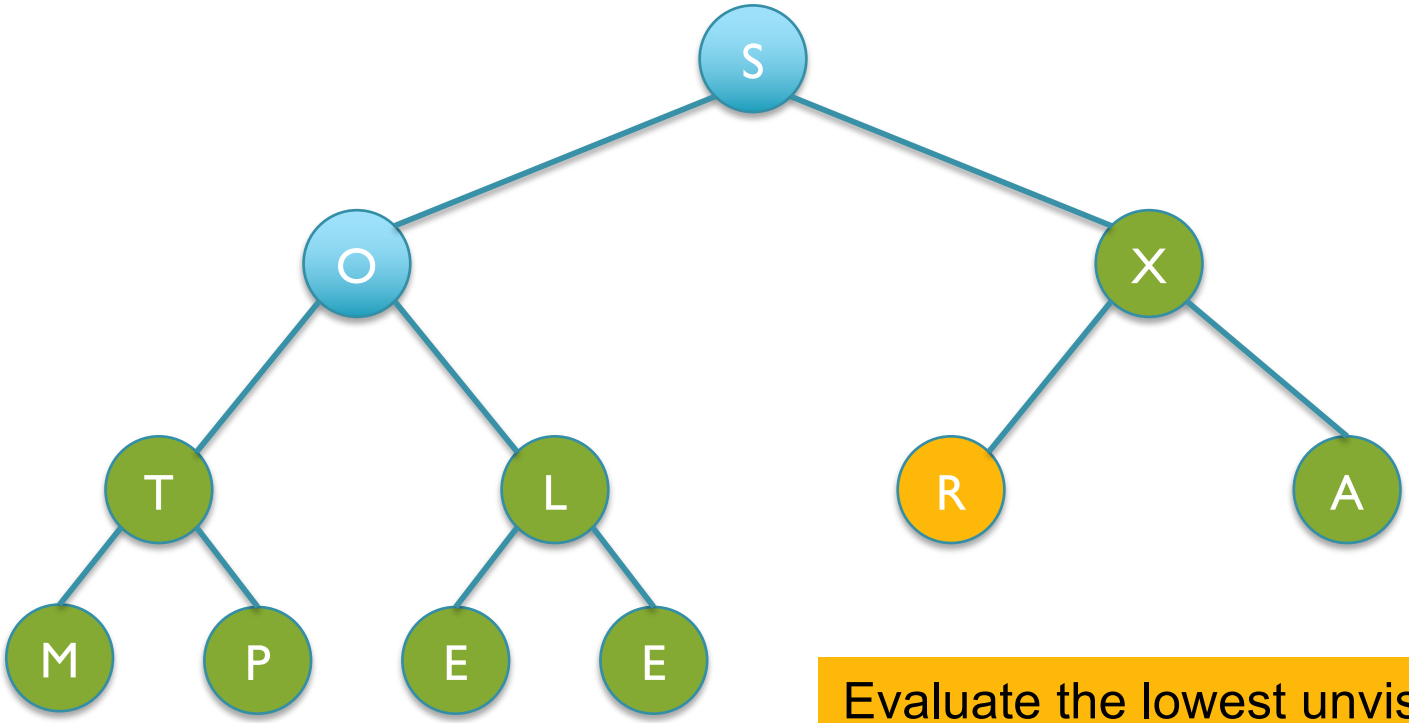


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	O	X	T	L	R	A	M	P	E	E

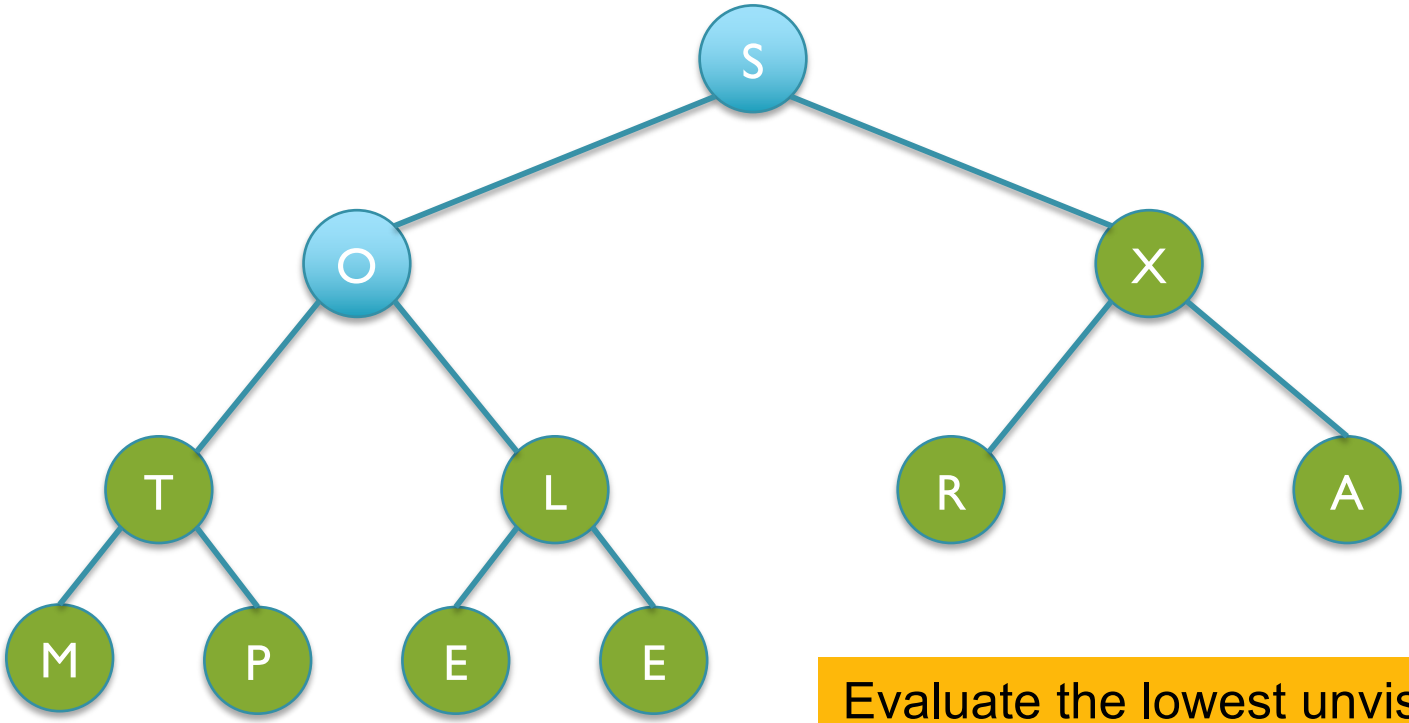


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	O	X	T	L	R	A	M	P	E	E

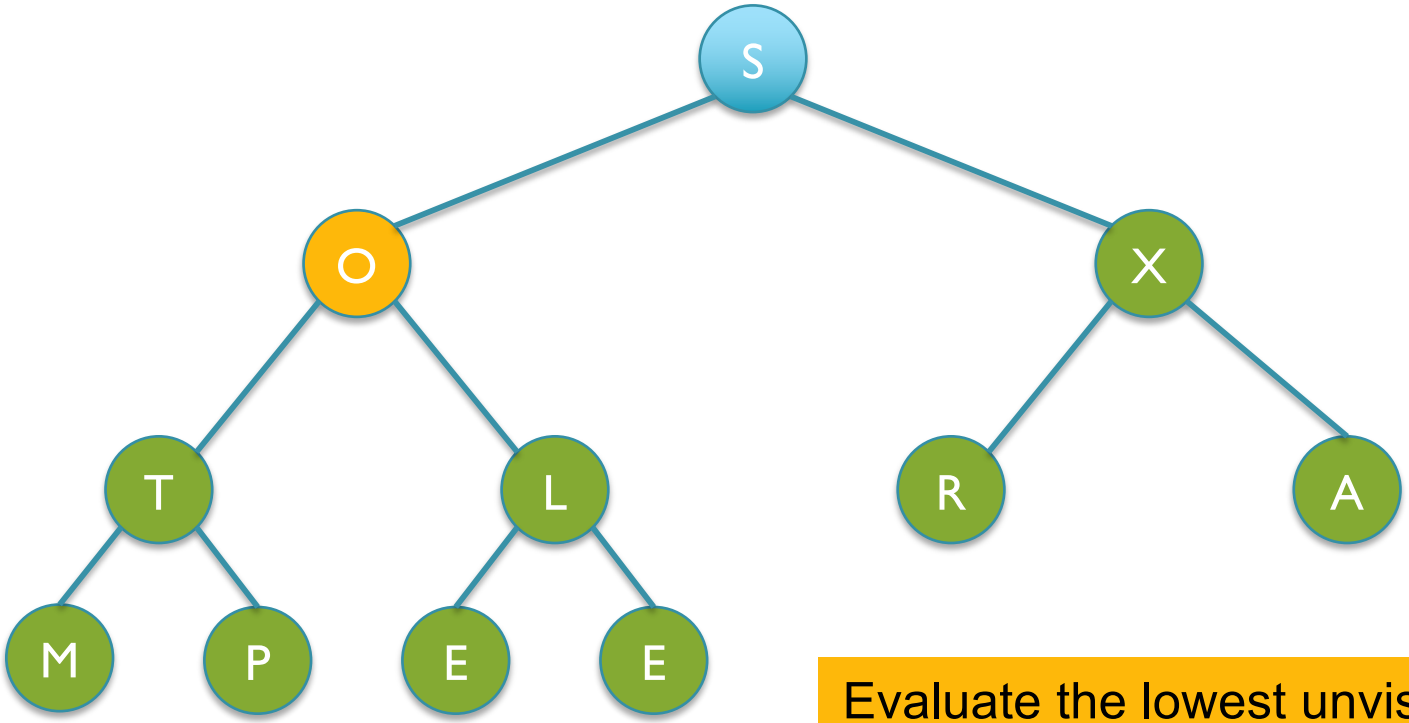


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	O	X	T	L	R	A	M	P	E	E

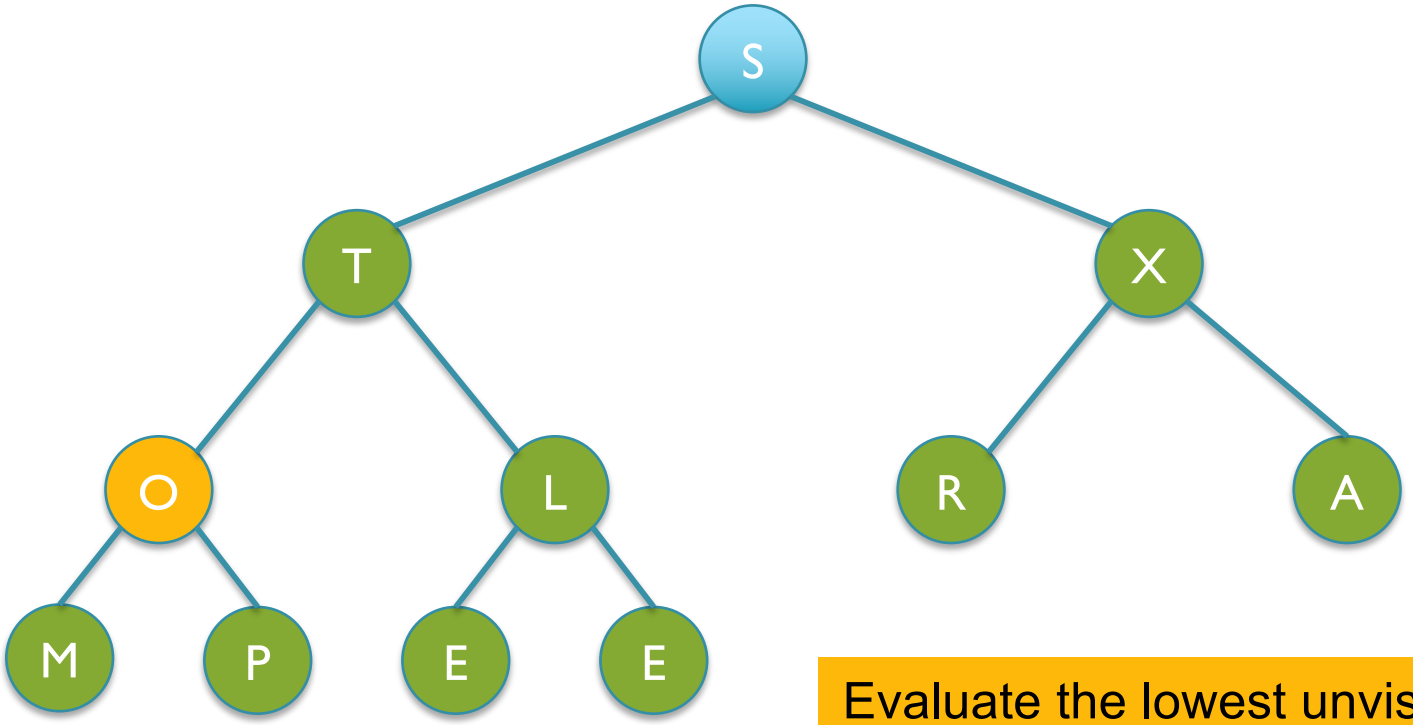


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	T	X	O	L	R	A	M	P	E	E

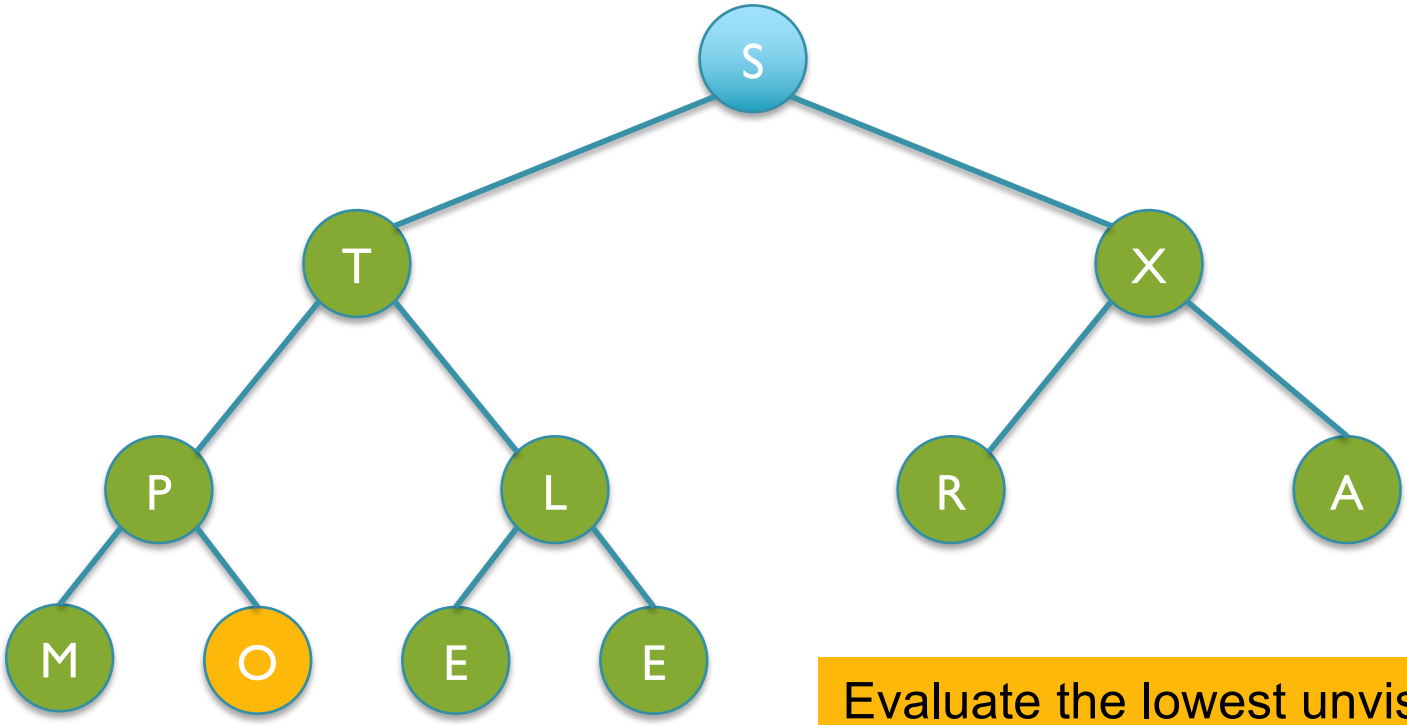


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively "heapify" all the nodes from the bottom up

S	T	X	P	L	R	A	M	O	E	E

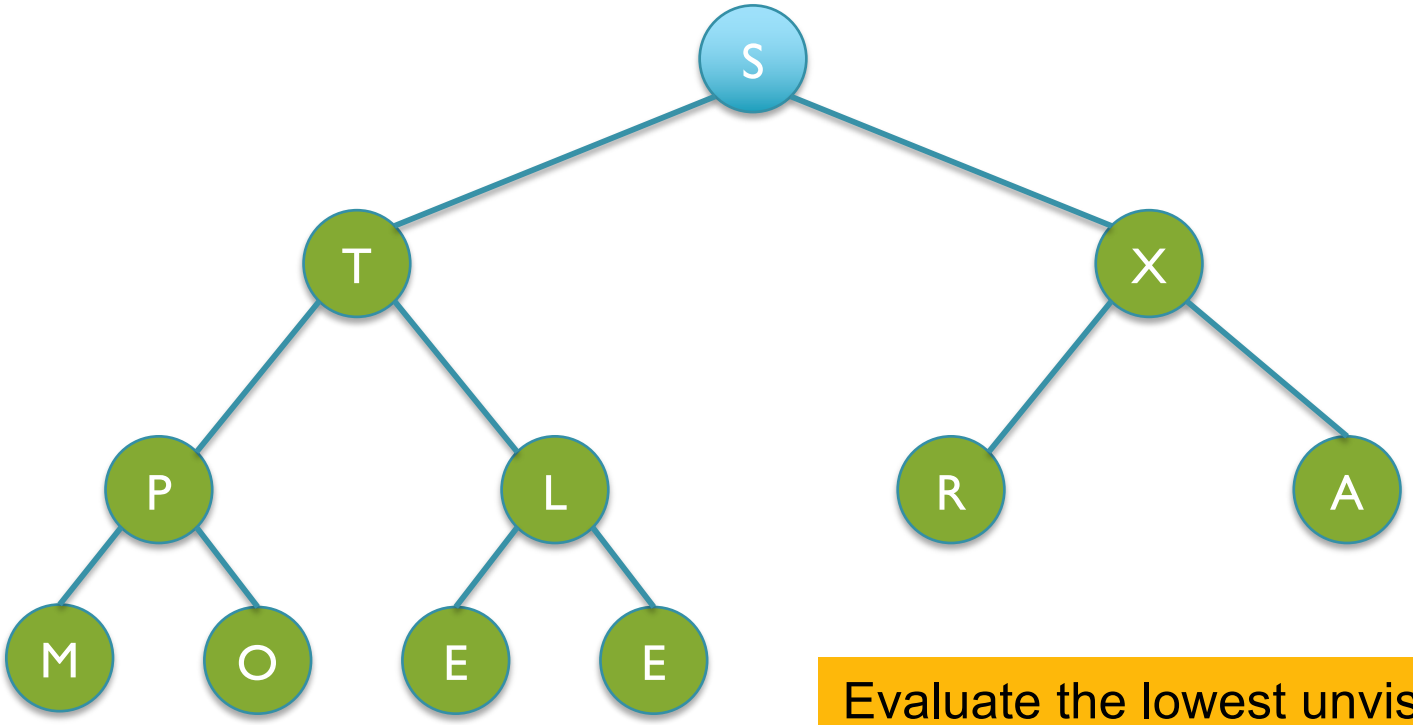


Evaluate the lowest unvisited node,
And "sift down" as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively "heapify" all the nodes from the bottom up

S	T	X	P	L	R	A	M	O	E	E

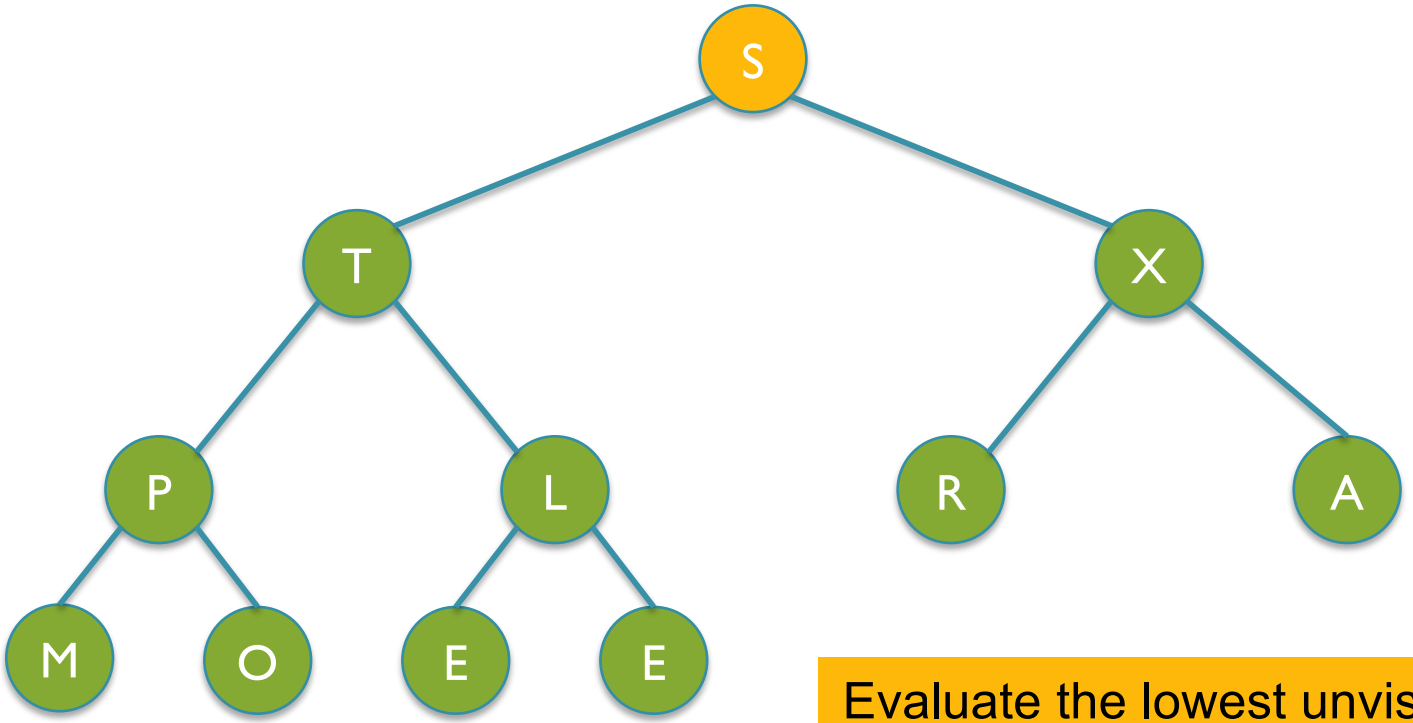


Evaluate the lowest unvisited node,
And "sift down" as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

S	T	X	P	L	R	A	M	O	E	E

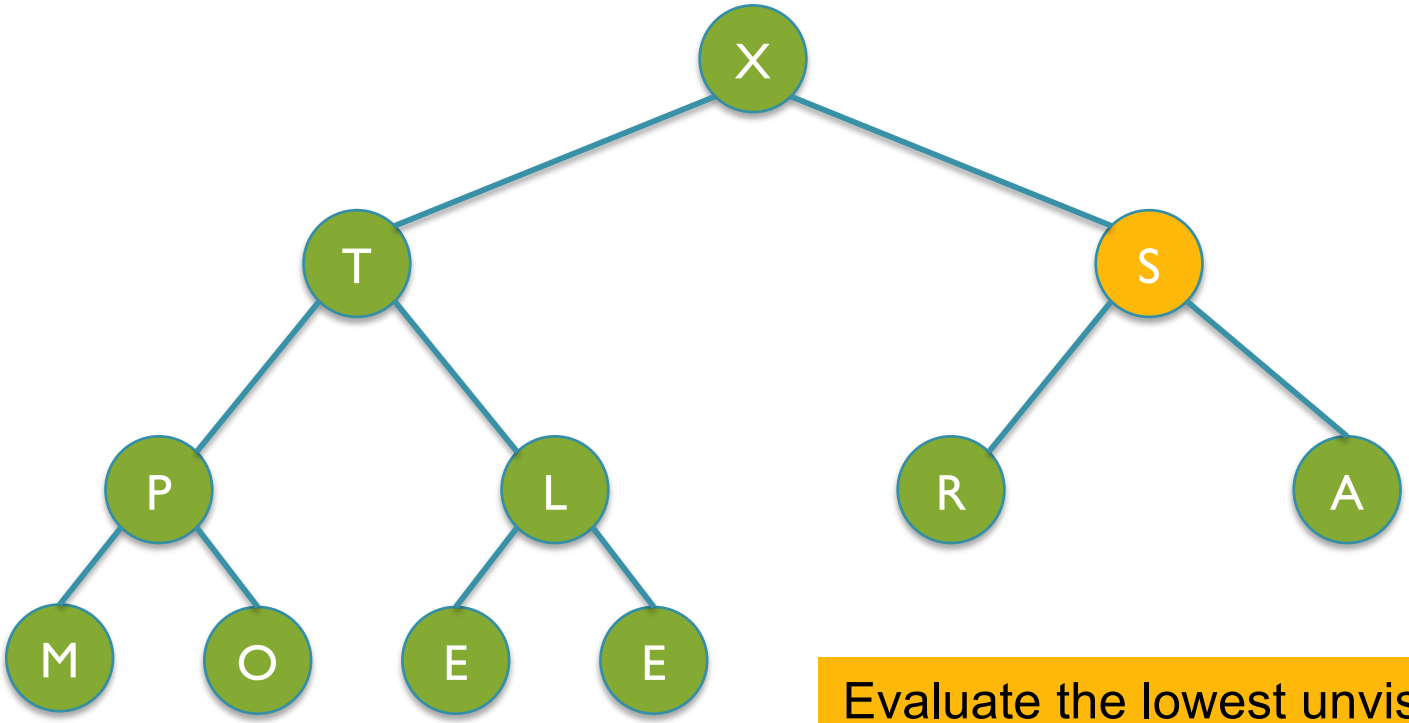


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

X	T	S	P	L	R	A	M	O	E	E

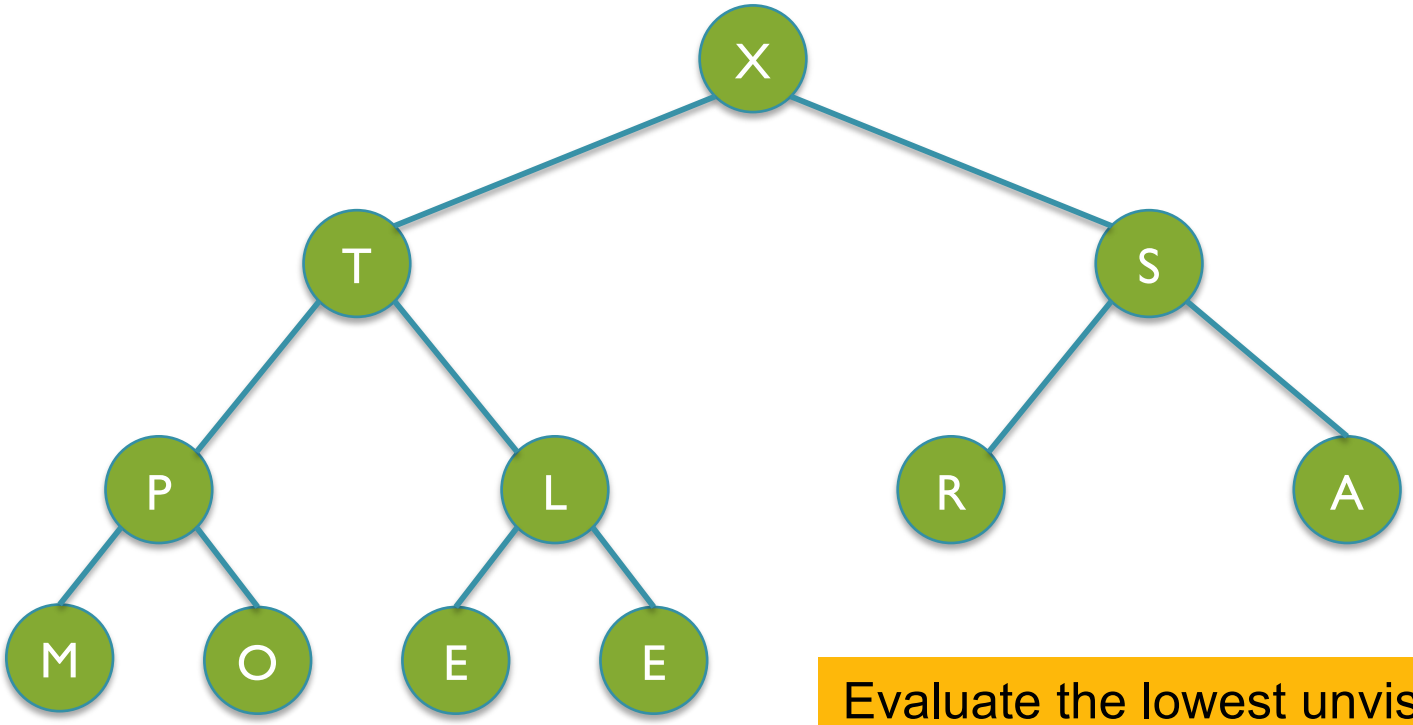


Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

Interestingly, it is possible to construct a heap from an unsorted array in O(N)
By iteratively “heapify” all the nodes from the bottom up

X	T	S	P	L	R	A	M	O	E	E



Evaluate the lowest unvisited node,
And “sift down” as needed

Bottom up Heapification

```
// Heapify Pseudo-code
// Note nodes n/2 through n are valid 1 node heaps to start ☺
// array starts at 1 so we can use the parent/child arithmetic

for (int i = n/2; i >= 1; i--) {
  siftdown(i)
}
```

A simple analysis would suggest O(n lg n) heapification:
For n/2 nodes, we need to siftdown by at most lg n steps.

But a more careful analysis shows that it is really O(n):

Height starts at 0 at the bottom, increases towards root

number of nodes at height h is $\leq \frac{2^{\lfloor \log n \rfloor}}{2^h} \leq \frac{n}{2^h}$

Each node at height h takes O(h) time
So overall runtime is:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

$$= O(n)$$

Note: $\sum_{i=0}^{\infty} i/2^i = 2$

Bottom up Heapification

Analysis of : $\sum_{i=0}^{\infty} i/2^i = 2$

Knowing that:

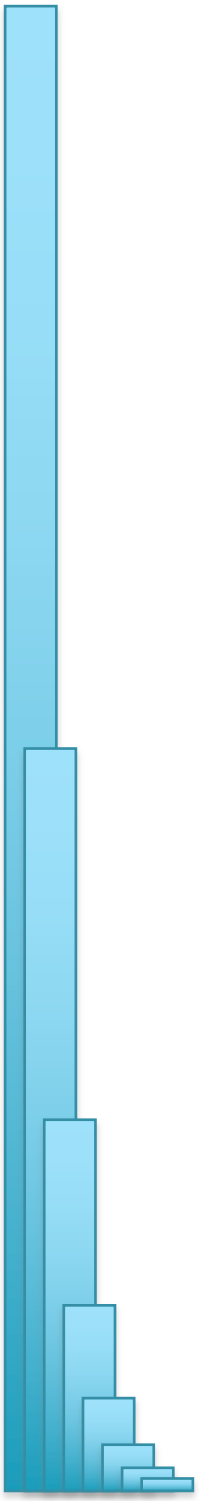
$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 1$$

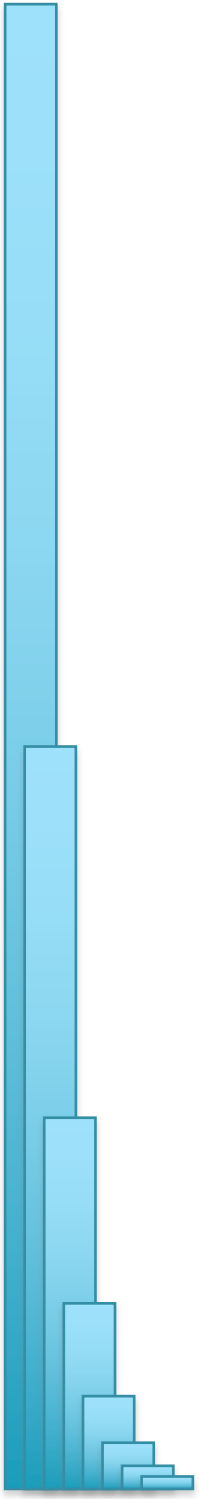
We have:

$$\begin{aligned}
\sum_{i=1}^{\infty} \frac{i}{2^i} &= \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots \\
&= (\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots) + (\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots) + (\frac{1}{8} + \frac{1}{16} + \cdots) + \cdots \\
&= 1 + \frac{1}{2} + \frac{1}{4} + \cdots \\
&= 2
\end{aligned}$$

Agenda

1. ***Recap on BitSets***
2. ***Maps***
3. ***BSTs***





Part 1. 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

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 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:

$$101010_2 = 1x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 0x2^0$$
$$= 1x32 + 0x16 + 1x8 + 0x4 + 1x2 + 0x1$$
$$= 32_{10} + 0_{10} + 8_{10} + 0_{10} + 2_{10} + 0_{10}$$
$$= 42_{10}$$

Decimal:

$$4711_{10} = 4x10^3 + 7x10^2 + 1x10^1 + 1x10^0$$
$$= 4x1000 + 7x100 + 1x10 + 1x1$$
$$= 4000 + 700 + 10 + 1$$

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

```
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: " +
}
```

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()

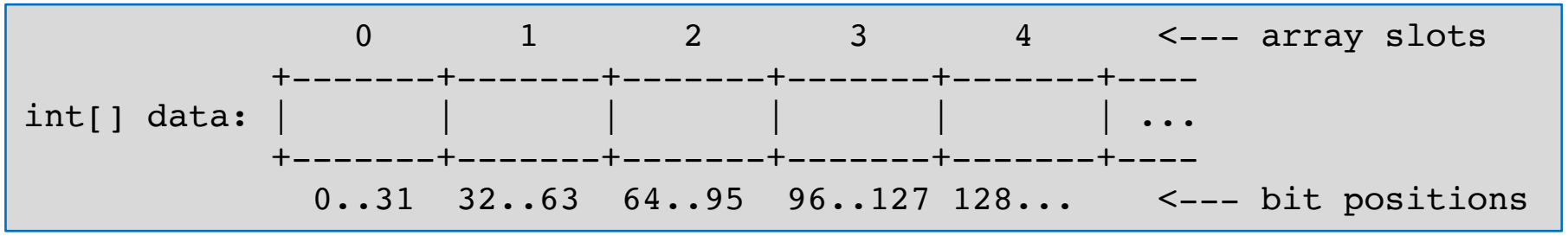
```
iset = 0
```

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 ☺



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 ☺

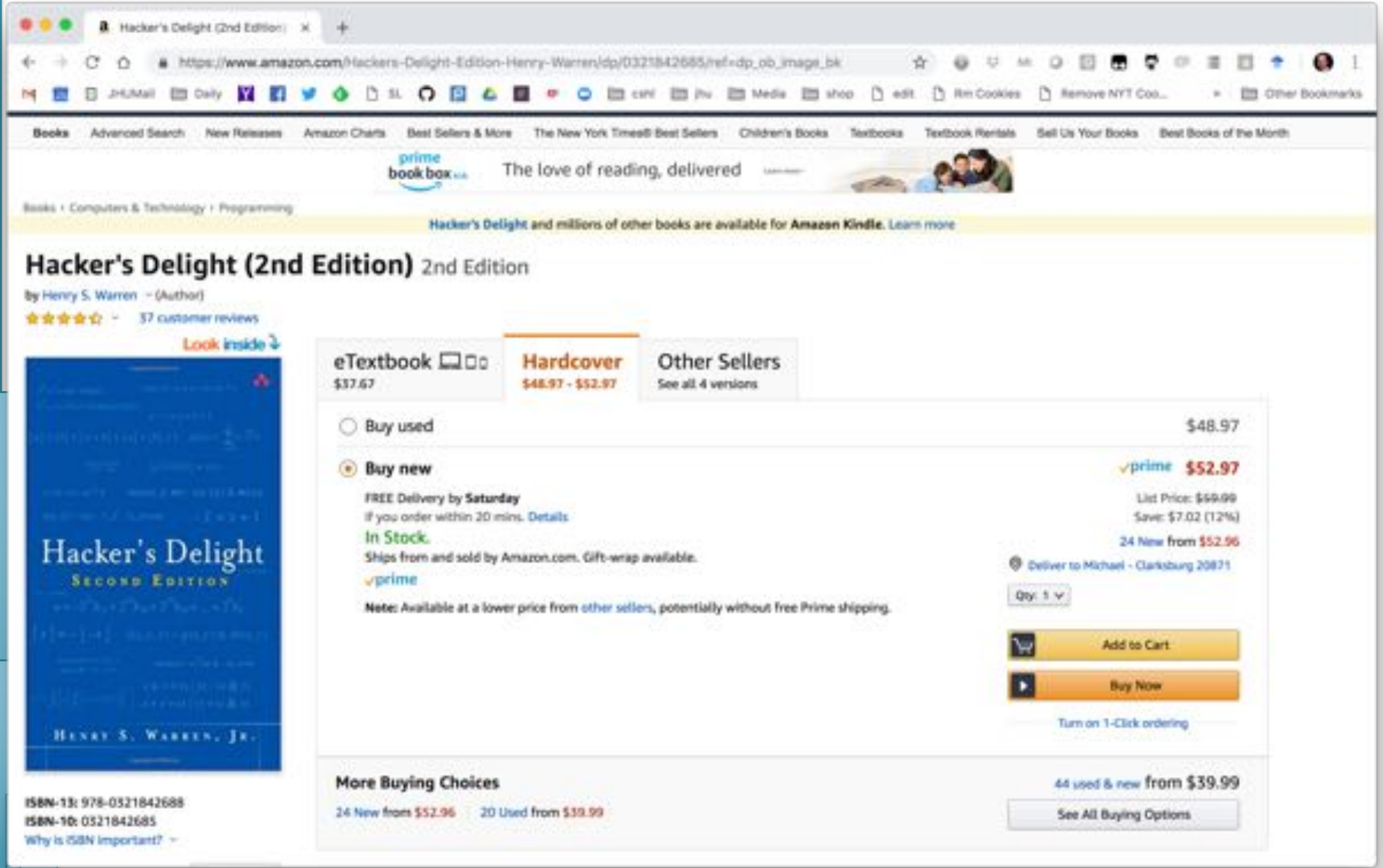
Bit Twiddling Hacks

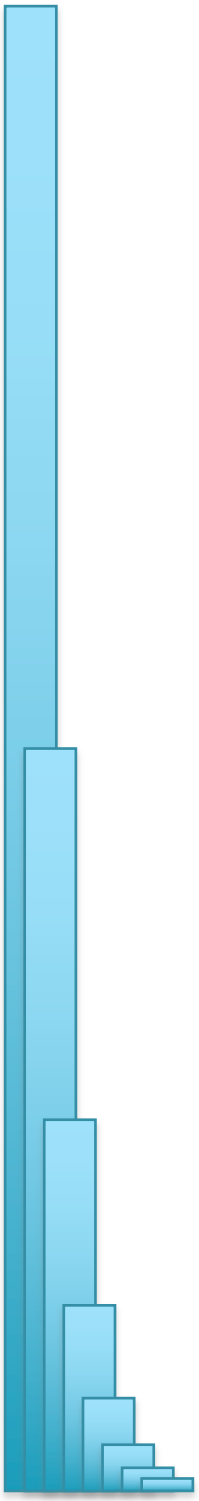
https://graphics.stanford.edu/~seander/bithacks.html

Contents

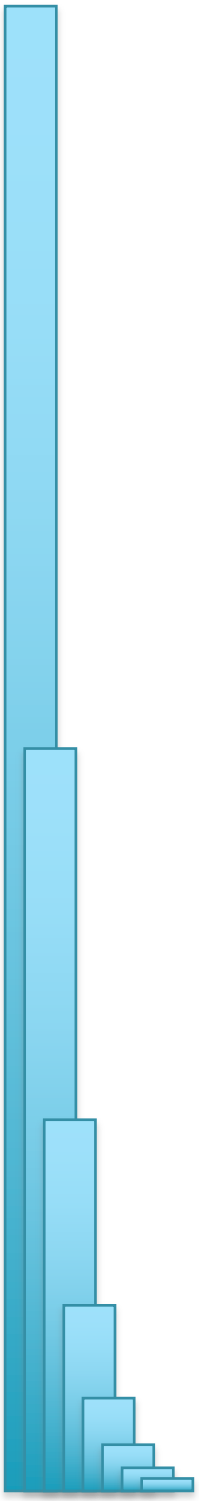
- About the operation counting methodology
- Compute the sign of an integer
- Detect if two integers have opposite signs
- Compute the integer absolute value (abs) without branching
- Compute the minimum (min) or maximum (max) of two integers without branching
- Determining if an integer is a power of 2
- Sign extending
  - Sign extending from a constant bit-width
  - Sign extending from a variable bit-width
  - Sign extending from a variable bit-width in 3 operations
- Conditionally set or clear bits without branching
- Conditionally negate a value without branching
- Merge bits from two values according to a mask
- Counting bits set
  - Counting bits set, naive way
  - Counting bits set by lookup table
  - Counting bits set, Brian Kernighan's way
  - Counting bits set in 14, 24, or 32-bit words using 64-bit instructions
  - Counting bits set, in parallel
  - Count bits set (rank) from the most-significant bit upto a given position
  - Select the bit position (from the most-significant bit) with the given count (rank)
- Computing parity (1 if an odd number of bits set, 0 otherwise)
  - Compute parity of a word the naive way
  - Compute parity by lookup table
  - Compute parity of a byte using 64-bit multiply and modulus division
  - Compute parity of word with a multiply
  - Compute parity in parallel
- Swapping Values
  - Swapping values with subtraction and addition
  - Swapping values with XOR
  - Swapping individual bits with XOR
- Reversing bit sequences
  - Reverse bits the obvious way
  - Reverse bits in word by lookup table
  - Reverse the bits in a byte with 3 operations (64-bit multiply and modulus division)
  - Reverse the bits in a byte with 4 operations (64-bit multiply, no division)
  - Reverse the bits in a byte with 7 operations (no 64-bit, only 32)
  - Reverse an N-bit quantity in parallel with 5 * lg(N) operations
- Modulus division (aka computing *remainders*)
  - Computing modulus division by 1 << s without a division operation (obvious)
  - Computing modulus division by (1 << s) - 1 without a division operation
  - Computing modulus division by (1 << s) - 1 in parallel without a division operation
- Finding integer log base 2 of an integer (aka the position of the highest bit set)
  - Find the log base 2 of an integer with the MSB N set in O(N) operations (the obvious way)
  - Find the integer log base 2 of an integer with an 64-bit IEEE float
  - Find the log base 2 of an integer with a lookup table
  - Find the log base 2 of an N-bit integer in O(lg(N)) operations
  - Find the log base 2 of an N-bit integer in O(lg(N)) operations with multiply and lookup
- Find integer log base 10 of an integer
- Find integer log base 10 of an integer the obvious way
- Find integer log base 2 of a 32-bit IEEE float
- Find integer log base 2 of the pow(2, r)-root of a 32-bit IEEE float (for unsigned integer r)
- Counting consecutive trailing zero bits (or finding bit indices)
  - Count the consecutive zero bits (trailing) on the right linearly
  - Count the consecutive zero bits (trailing) on the right in parallel
  - Count the consecutive zero bits (trailing) on the right by binary search
  - Count the consecutive zero bits (trailing) on the right by casting to a float
  - Count the consecutive zero bits (trailing) on the right with modulus division and lookup
  - Count the consecutive zero bits (trailing) on the right with multiply and lookup
- Round up to the next highest power of 2 by float casting
- Round up to the next highest power of 2
- Interleaving bits (aka computing *Morton Numbers*)
  - Interleave bits the obvious way
  - Interleave bits by table lookup
  - Interleave bits with 64-bit multiply
  - Interleave bits by Binary Magic Numbers
- Testing for ranges of bytes in a word (and counting occurances found)
  - Determine if a word has a zero byte
  - Determine if a word has a byte equal to n
  - Determine if a word has byte less than n
  - Determine if a word has a byte greater than n
  - Determine if a word has a byte between m and n
- Compute the lexicographically next bit permutation

Hacker's Delight





Part 2:Maps



Maps

aka dictionaries

aka associative arrays

```
Mike      -> Malone 323
Peter     -> Malone 223
Joanne     -> Malone 225
Zack      -> Malone 160 suite
Debbie     -> Malone 160 suite
Randal     -> Malone 160 suite
Ron       -> Garland 242
```

Key (of Type K) -> Value (of Type V)

Note you can have multiple keys with the same value,
But not okay to have one key map to more than 1 value

How might you map to more than 1 value?

Maps, Sets, and Arrays

Sets as Map<T, Boolean>

```
Mike      -> True
Peter     -> True
Joanne    -> True
Zack      -> True
Debbie    -> True
Yair      -> True
Ron       -> True
```

Array as Map<Integer, T>

```
0   -> Mike
1   -> Peter
2   -> Joanne
3   -> Zack
4   -> Debbie
5   -> Yair
6   -> Ron
```

Maps are extremely flexible and powerful,
and therefore are extremely widely used

Built into many common languages: Awk, Python, Perl, JavaScript…

How could maps be used with sparse arrays?

How could maps be used with graphs?

Map Interface

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

    void insert (K k, V v);
    void remove(K k);
    V get(K k);

}
```

What functionality does “extends Iterable<K> introduce?

What should get() return if k is not in the map?

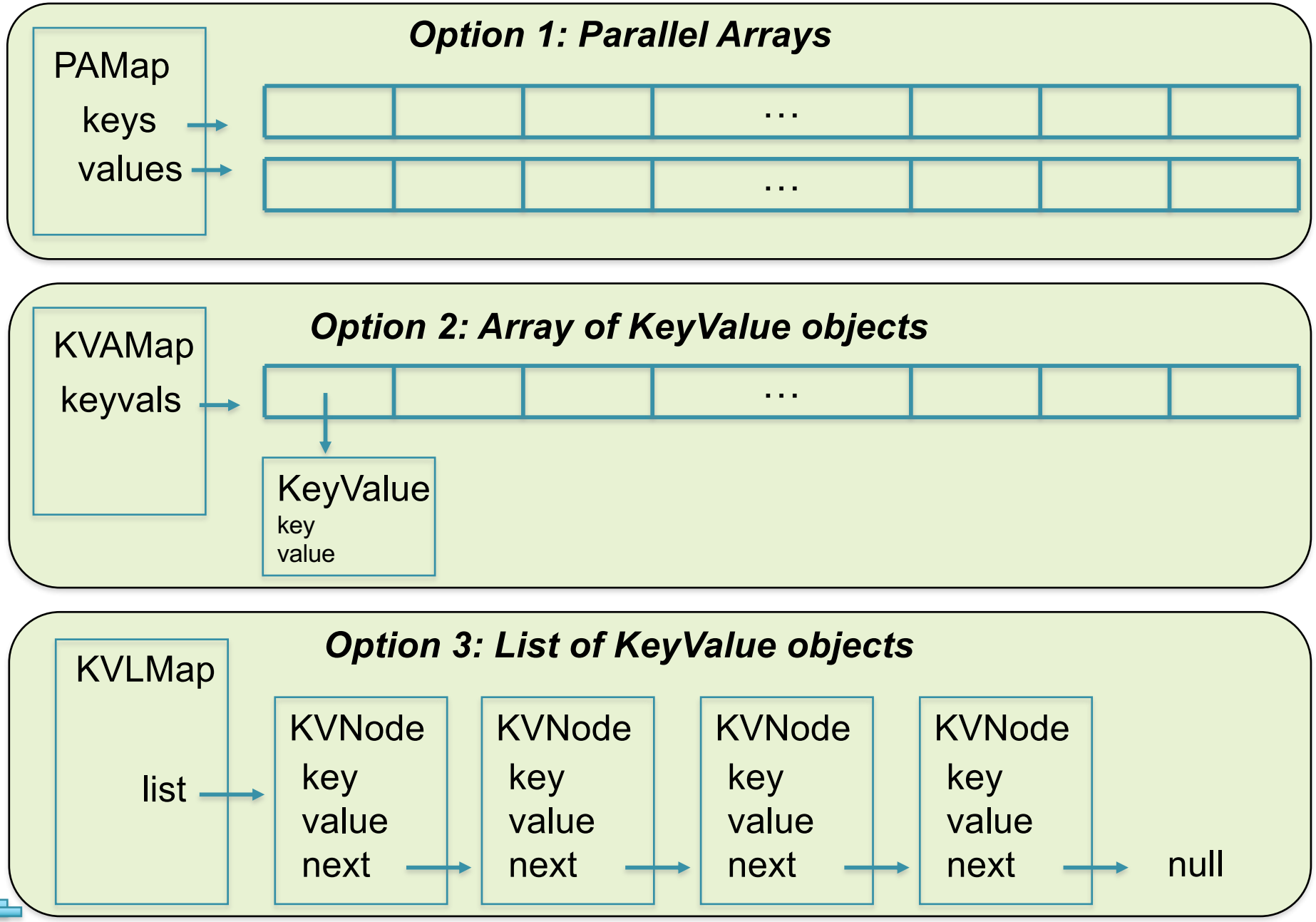
What should insert() do if k is already in the map?

Map Interface v2

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

    void insert (K k, V v) throws DuplicateKeyException;
    V remove(K k) throws UnknownKeyException;
    void put(K k, V v) throws UnknownKeyException;
    V get(K k) throws UnknownKeyException;
    boolean has(K k);

}
```

How would you implement this interface?

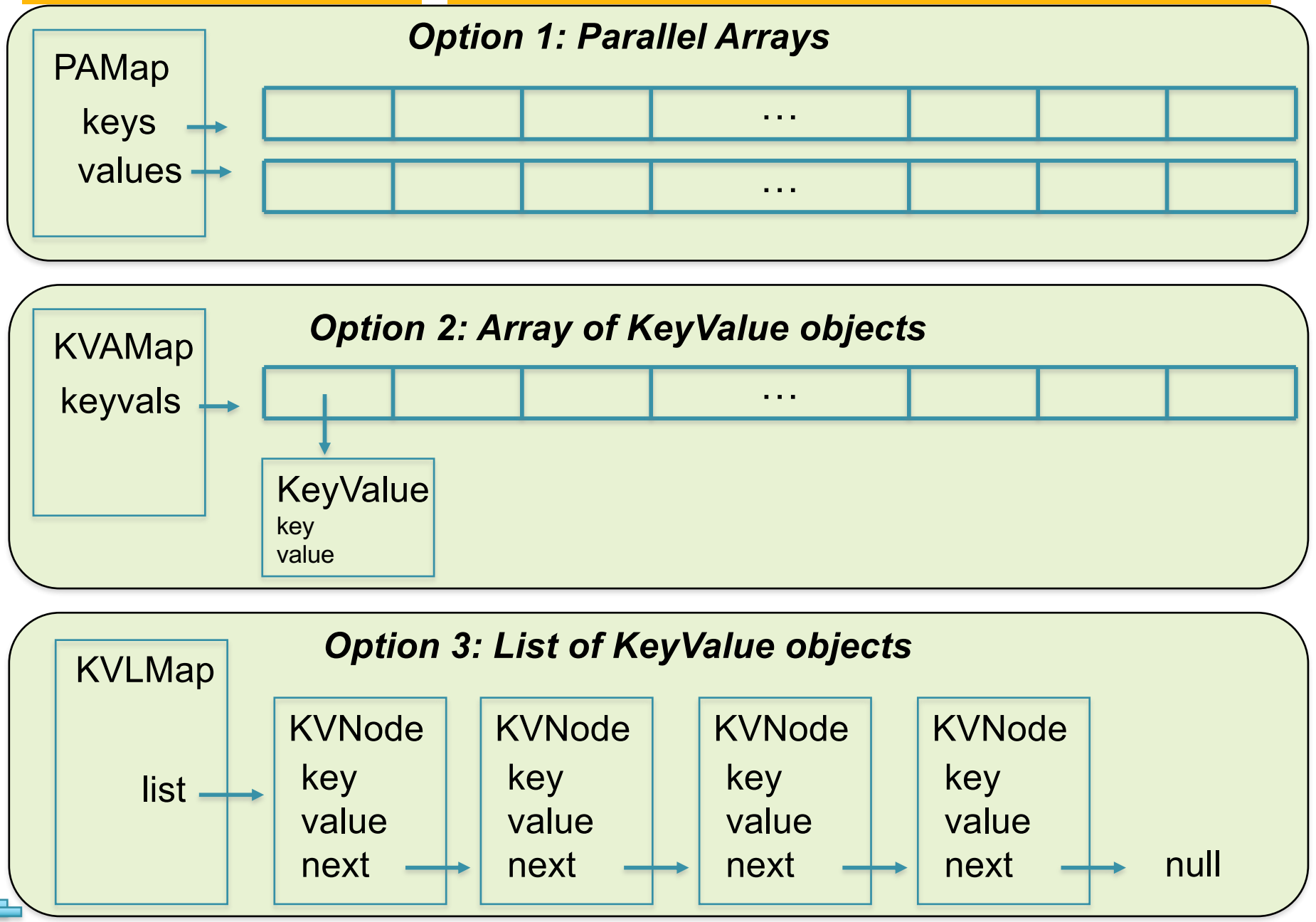
Map Implementation



Map Implementation

Can we do any better?

Yes, but only if the keys can be ordered / sorted!



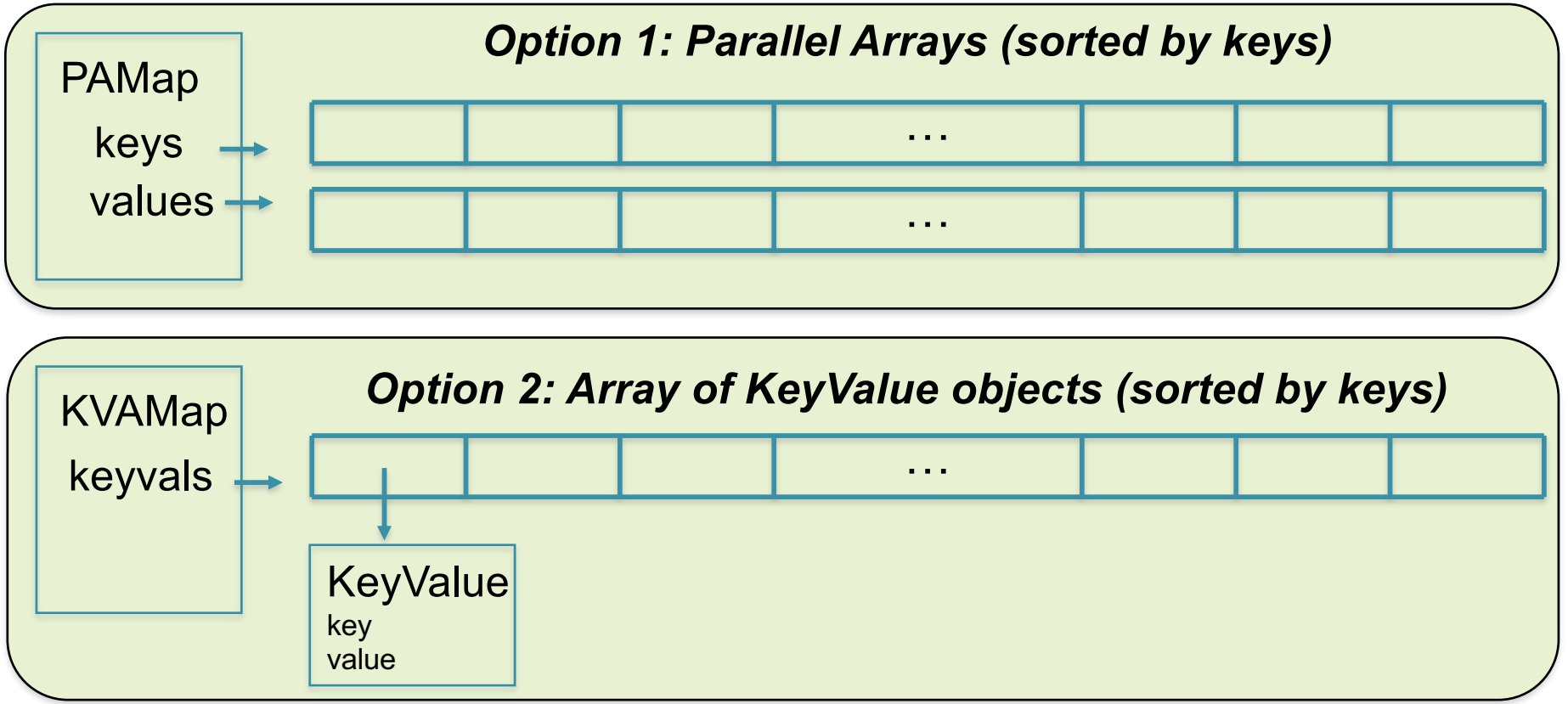
Map Interface v3

```
public interface OrderedMap<K extends Comparable<K>,V>
          extends Iterable<K>{

    void insert (K k, V v) throws DuplicateKeyException;
    V remove(K k) throws UnknownKeyException;
    void put(K k, V v) throws UnknownKeyException;
    V get(K k) throws UnknownKeyException;
    boolean has(K k);

}
```

Woohoo! Now keys can be compared to each other

How would you implement this interface?

Map Implementation



If the arrays are sorted, we can use binary search =>
get() and has() will run in O(lg n) time!

What about insert() or remove()?

O(n) time ☹

Could we do everything in O(lg n) time or faster?

Next Steps

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