CS 600.226: Data Structures

Michael Schatz

Nov 5, 2018

Lecture 28. Hash Tables

HW7

Assignment 7: Whispering Trees

Out on: November 2, 2018

Due by: November 9, 2018 before 10:00 pm

Collaboration: None

Grading:

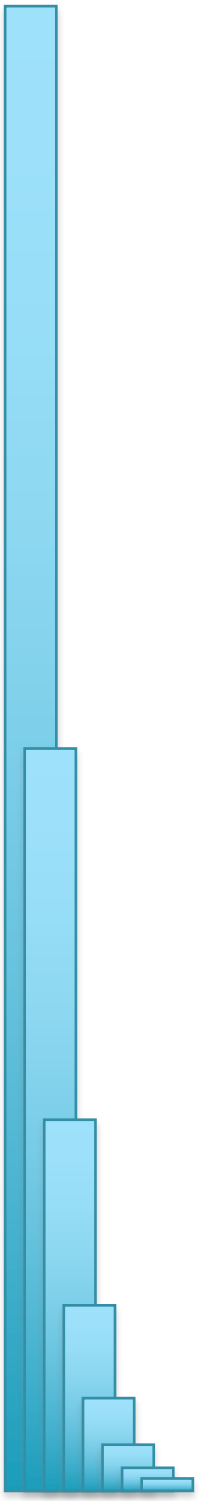
- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

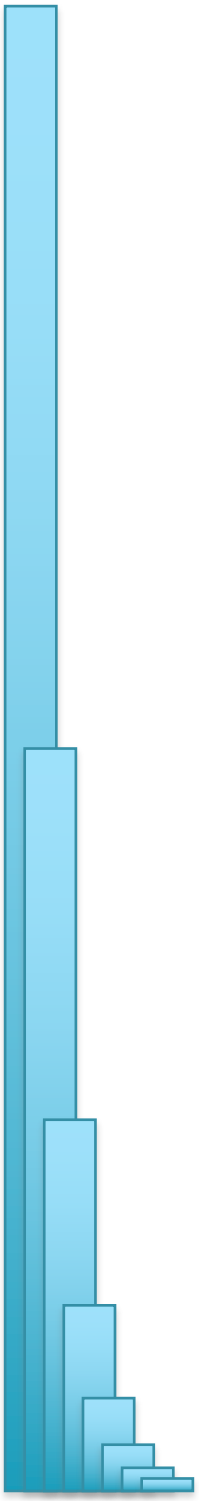
Overview

The seventh assignment is all about ordered maps, specifically fast ordered maps in the form of balanced binary search trees. You'll work with a little program called Words that reads text from standard input and uses an (ordered) map to count how often different words appear. We're giving you a basic (unbalanced) binary search tree implementation of OrderedMap that you can use to play around with the Words program and as starter code for your own developments.

Agenda

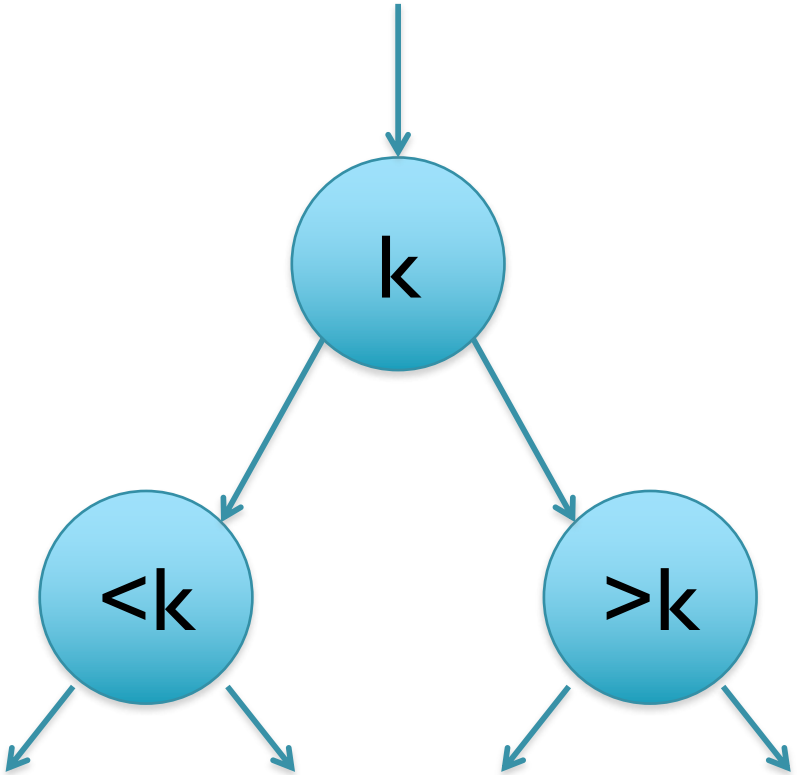
1. *Recap on BST, AVL trees and Treaps*
2. *Hash Tables*





Part 1: BST

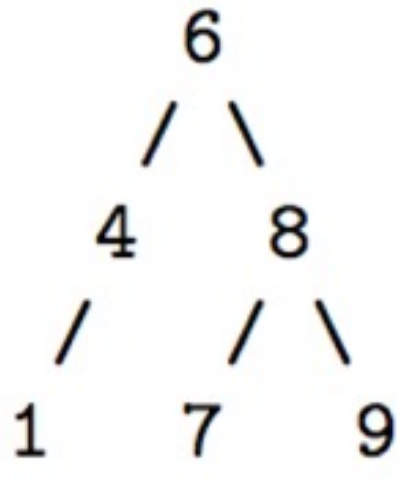
Binary Search Tree



A BST is a binary tree with a special ordering property:
If a node has value k, then the left child (and its descendants) will have values smaller than k; and the right child (and its descendants) will have values greater than k

Can a BST have duplicate values?

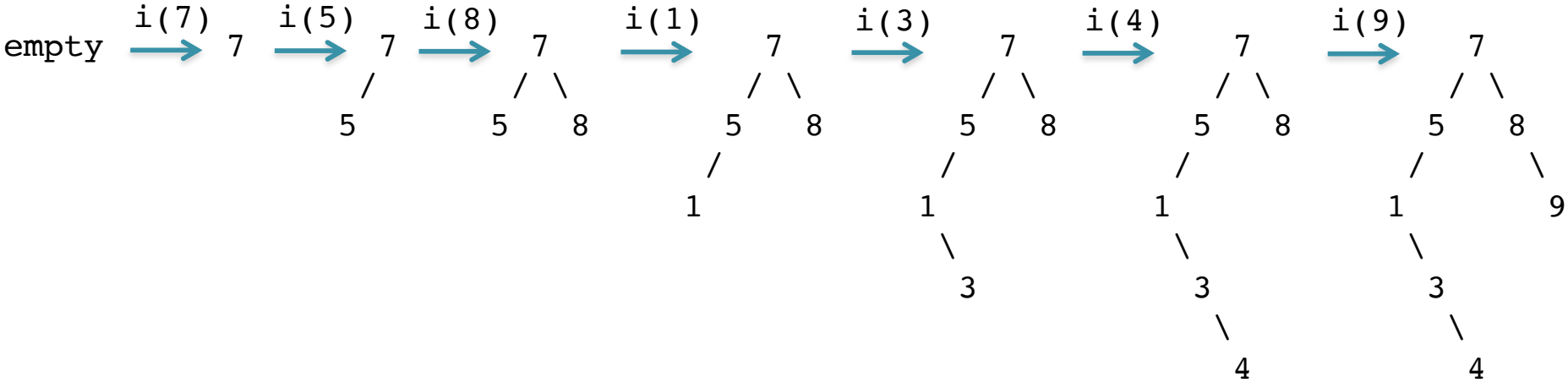
Searching



has(7):	compare 6 => compare 8 => found 7
has (2):	compare 6 => compare 4 => compare 1 => not found!

What is the runtime for has() ?

Constructing

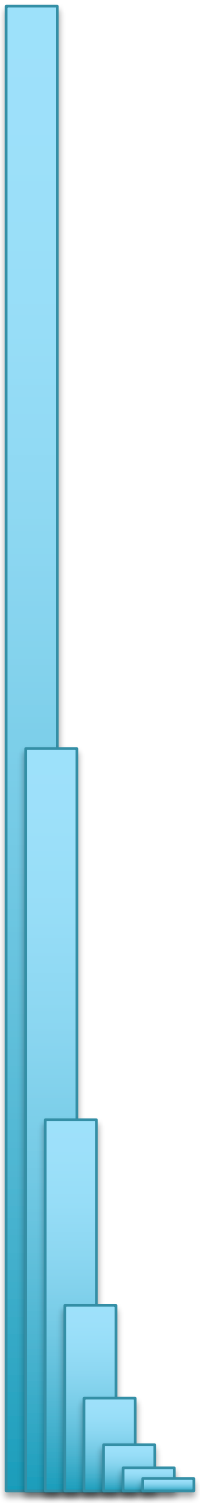


Note the shape of a general BST will depend on the order of insertions

What is the “worst” order for constructing the BST?

What is the “best” order for constructing the BST?

What happens for a random ordering?



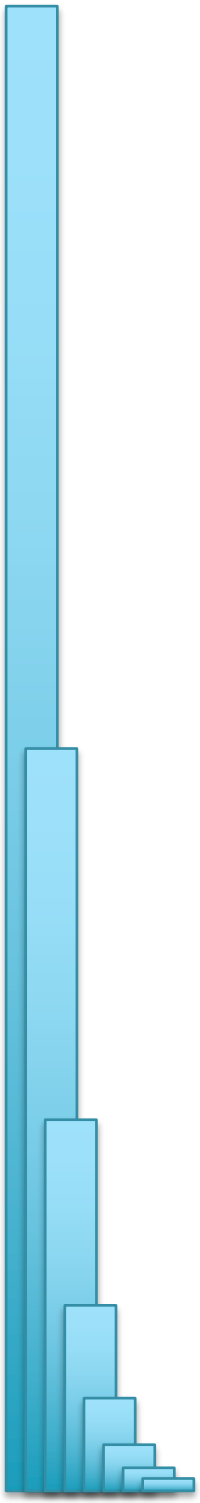
Random Tree Height

Trying all permutations of 14 distinct keys

numtrees: 87,178,291,200 average height: 6.63

maxheights[0]:	0	0.00%
maxheights[1]:	0	0.00%
maxheights[2]:	0	0.00%
maxheights[3]:	0	0.00%
maxheights[4]:	21964800	0.03%
maxheights[5]:	10049994240	11.53%
maxheights[6]:	33305510656	38.20%
maxheights[7]:	27624399104	31.69%
maxheights[8]:	12037674752	13.81%
maxheights[9]:	3393895680	3.89%
maxheights[10]:	652050944	0.75%
maxheights[11]:	85170176	0.10%
maxheights[12]:	7258112	0.01%
maxheights[13]:	364544	0.00%
maxheights[14]:	8192	0.00%

7hrs 17m



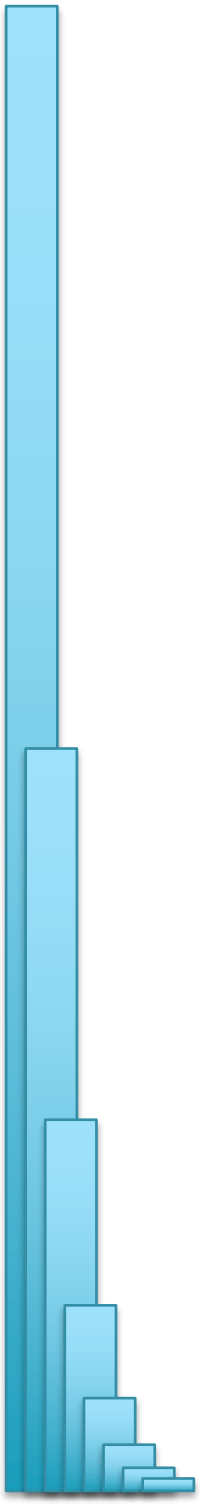
Random Tree Height

Trying all permutations of 15 distinct keys

numtrees: 1,307,674,368,000 average height: 6.83

maxheights[0]:	0	0.00%
maxheights[1]:	0	0.00%
maxheights[2]:	0	0.00%
maxheights[3]:	0	0.00%
maxheights[4]:	21964800	0.00%
maxheights[5]:	92644597760	7.08%
maxheights[6]:	450049847808	34.42%
maxheights[7]:	450900458496	34.48%
maxheights[8]:	223762187264	17.11%
maxheights[9]:	71589889024	5.47%
maxheights[10]:	15916504576	1.22%
maxheights[11]:	2496484352	0.19%
maxheights[12]:	271953920	0.02%
maxheights[13]:	19619840	0.00%
maxheights[14]:	843776	0.00%
maxheights[15]:	16384	0.00%

5 days 22hrs 11min

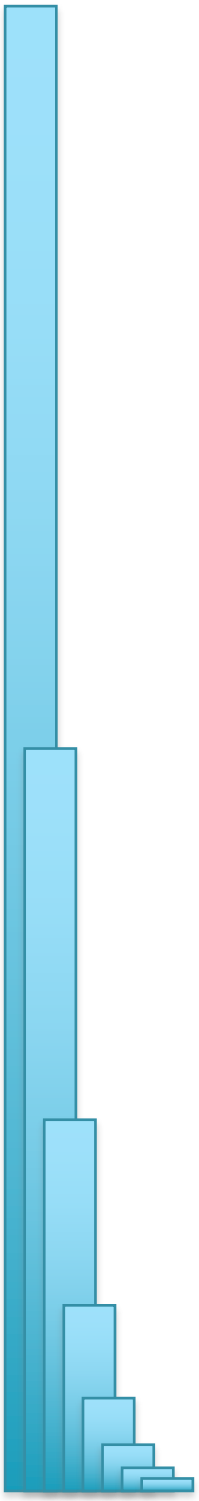


Random Tree Height

Trying all permutations of 16 distinct keys

$ tail -20 heights.16.log
tree[96520000000]: 1 3 4 10 11 6 2 7 8 14 9 13 15 16 12 5 maxheight: 8
tree[96521000000]: 1 3 4 10 11 6 13 8 9 2 7 15 14 12 5 16 maxheight: 8
tree[96522000000]: 1 3 4 10 11 6 15 14 7 13 2 12 5 8 9 16 maxheight: 9
tree[96523000000]: 1 3 4 10 11 7 8 12 13 16 6 9 15 2 5 14 maxheight: 10
tree[96524000000]: 1 3 4 10 11 7 5 2 12 9 6 15 14 8 13 16 maxheight: 9
tree[96525000000]: 1 3 4 10 11 7 14 8 2 15 5 12 13 6 9 16 maxheight: 8
tree[96526000000]: 1 3 4 10 11 7 16 15 9 2 6 13 8 5 12 14 maxheight: 9
tree[96527000000]: 1 3 4 10 11 8 9 12 15 13 7 5 14 6 2 16 maxheight: 9
tree[96528000000]: 1 3 4 10 11 8 12 2 13 16 15 7 9 14 5 6 maxheight: 10
tree[96529000000]: 1 3 4 10 11 8 15 6 12 9 14 13 7 16 2 5 maxheight: 9
tree[96530000000]: 1 3 4 10 11 9 7 15 2 8 13 6 14 12 5 16 maxheight: 8
tree[96531000000]: 1 3 4 10 11 9 2 13 6 5 8 12 7 14 15 16 maxheight: 9
tree[96532000000]: 1 3 4 10 11 9 13 2 15 14 12 7 6 16 5 8 maxheight: 8
tree[96533000000]: 1 3 4 10 11 9 16 8 14 2 5 15 6 12 13 7 maxheight: 9
tree[96534000000]: 1 3 4 10 11 2 8 15 12 13 5 9 6 14 7 16 maxheight: 9
tree[96535000000]: 1 3 4 10 11 2 5 13 6 15 16 8 9 7 12 14 maxheight: 8
…

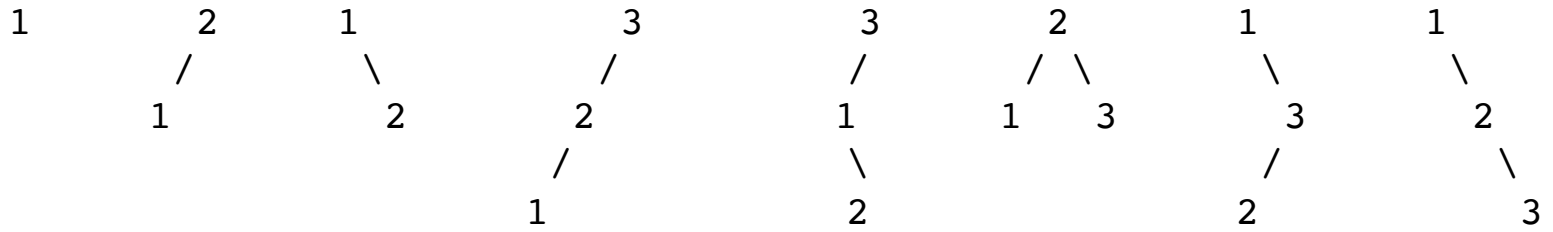
Only 95 more days to go ☺



Part 2: AVL Trees

Balanced Trees

Note that we cannot require a BST to be perfectly balanced:



Balanced

Not Balanced

Not Balanced

Not Balanced

Not Balanced

Balanced

Not Balanced

Not Balanced

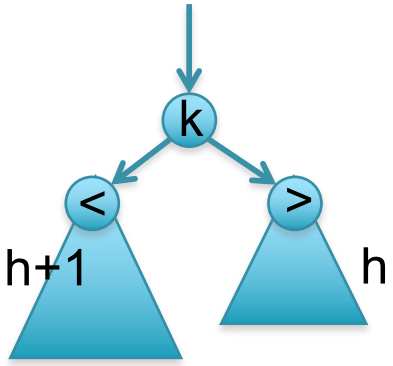
Since neither tree with 2 keys is perfectly balanced we cannot insist on it

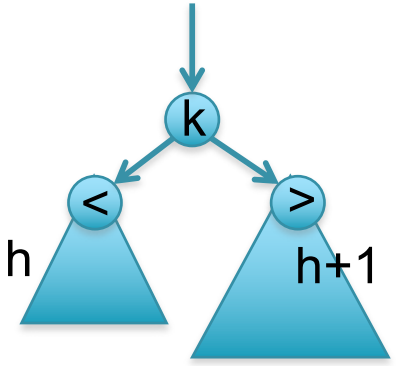
AVL Condition:

For every node n, the height of n's left and right subtree's differ by at most 1

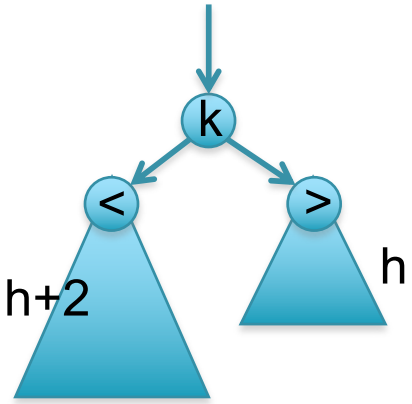
Maintaining Balance

Assume that the tree starts in a slightly unbalanced state:

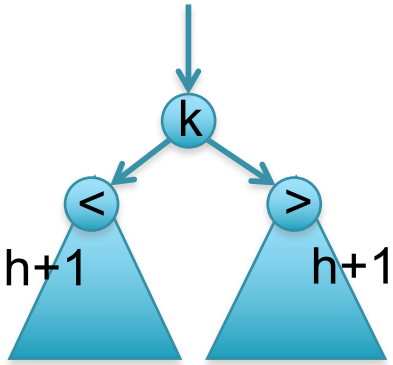




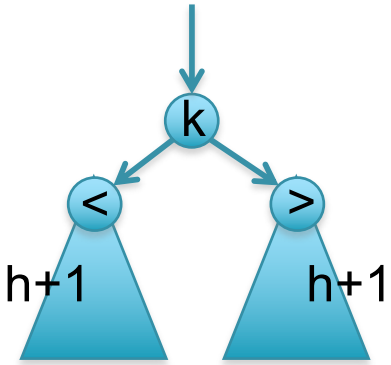
Inserting a new value can maintain the subtree heights or 1 of 4 possible outcomes:



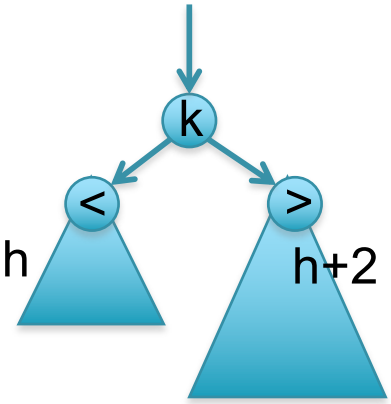
Insert left,
AVL Violation!



Insert right,
Rebalanced



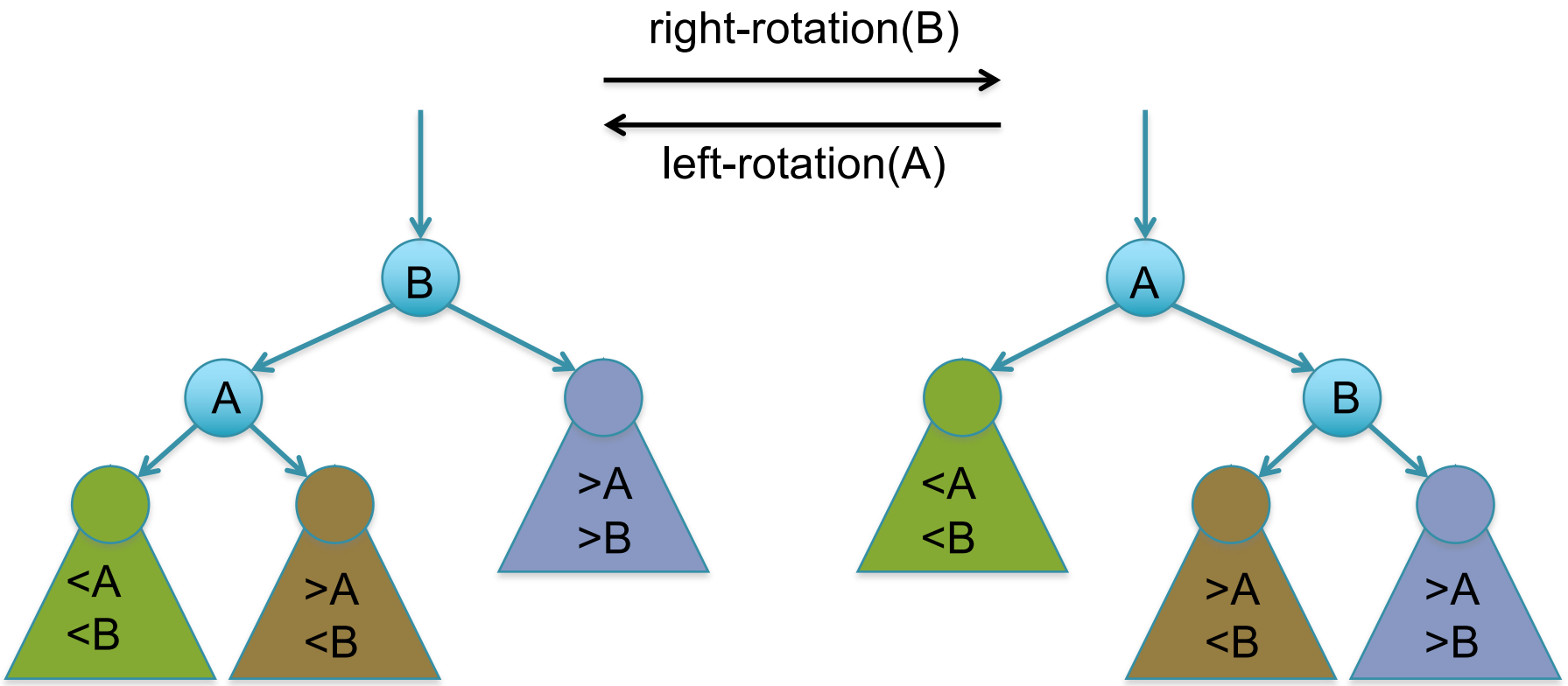
Insert left,
Rebalanced



Insert right,
AVL Violation!

Woohoo! AVL Violations ;-)

Tree Rotations

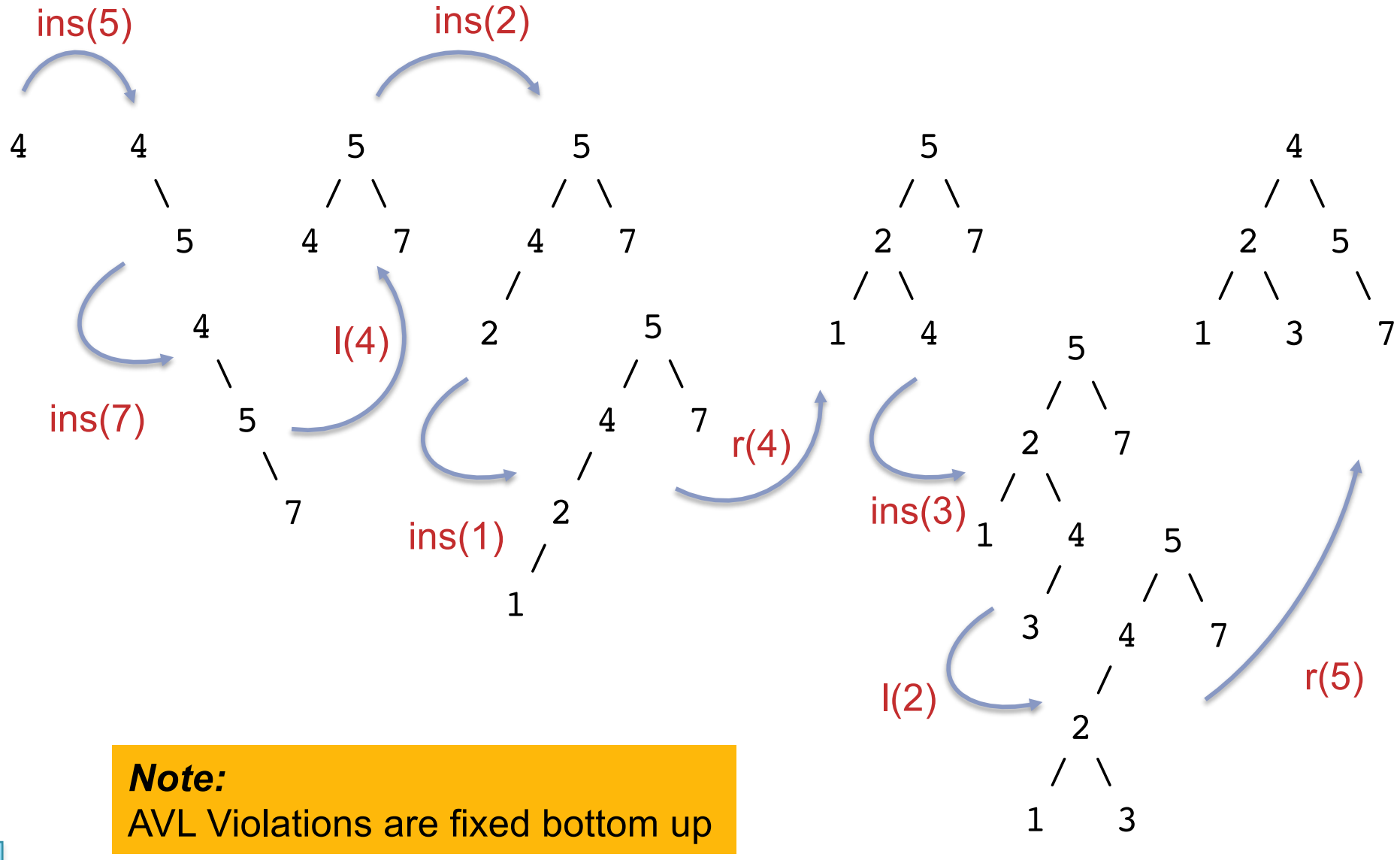


Note: Rotating a BST maintains the BST condition!

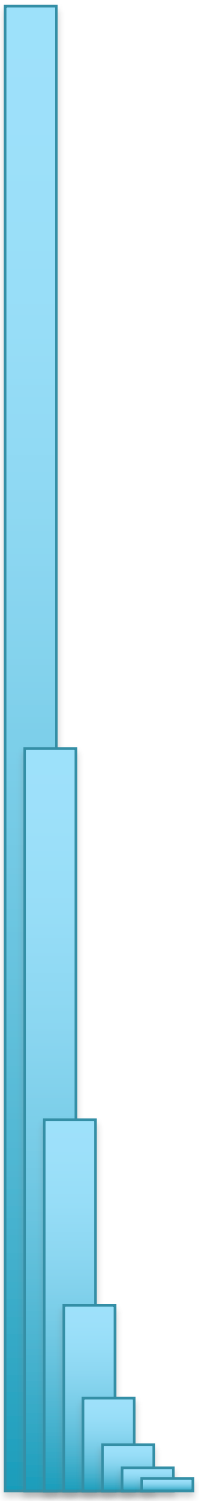
Green < A < Brown < B < Purple

Complete Example

Insert these values: 4 5 7 2 1 3

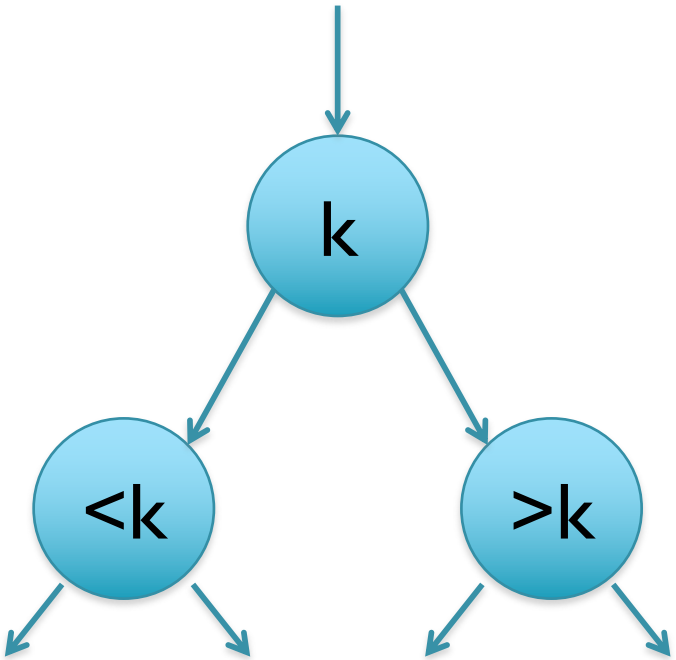


Note:
AVL Violations are fixed bottom up



Part 3: Treaps

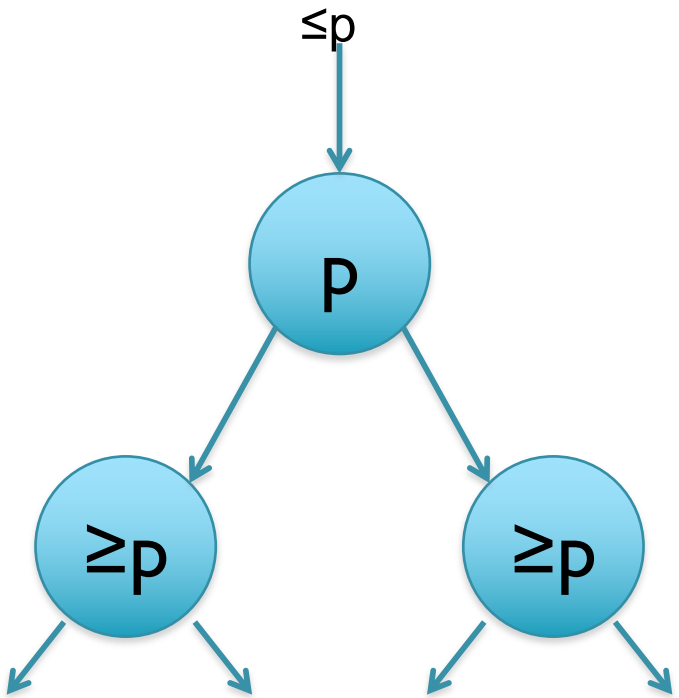
BSTs versus Heaps



BST

All keys in left subtree of k are < k, all keys in right are >k

Tricky to balance, but fast to find

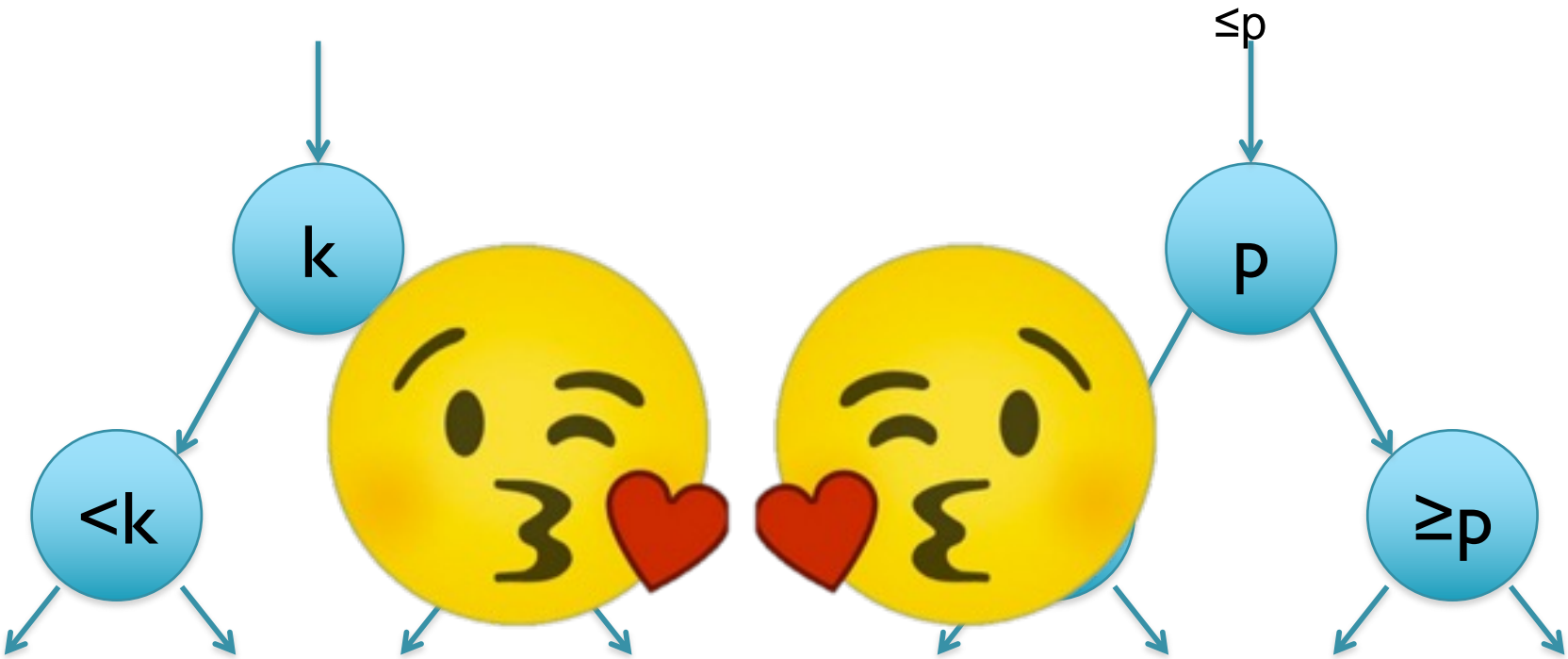


Heap

All children of the node with priority p have priority ≥p

Easy to balance, but hard to find (except min/max)

BSTs versus Heaps



BST

All keys in left subtree of k are < k, all keys in right are >k

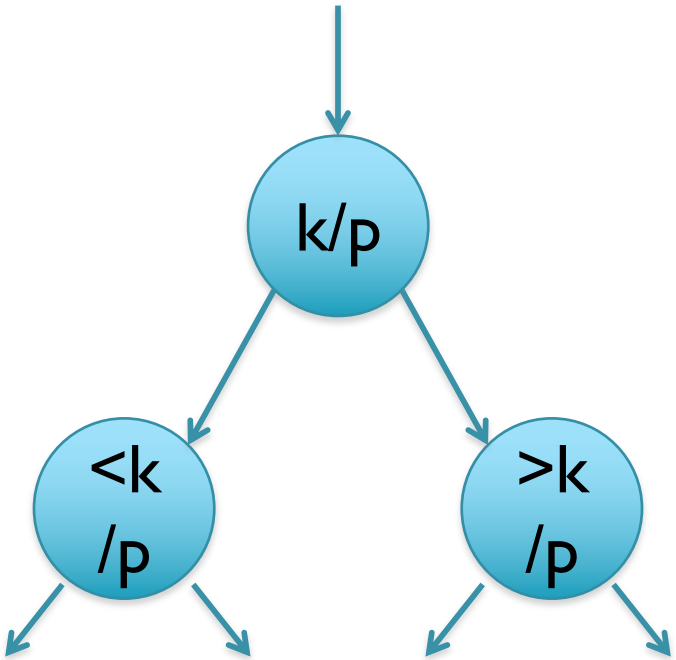
Tricky to balance, but fast to find

Heap

All children of the node with priority p have priority ≥p

Easy to balance, but hard to find (except min/max)

Treap

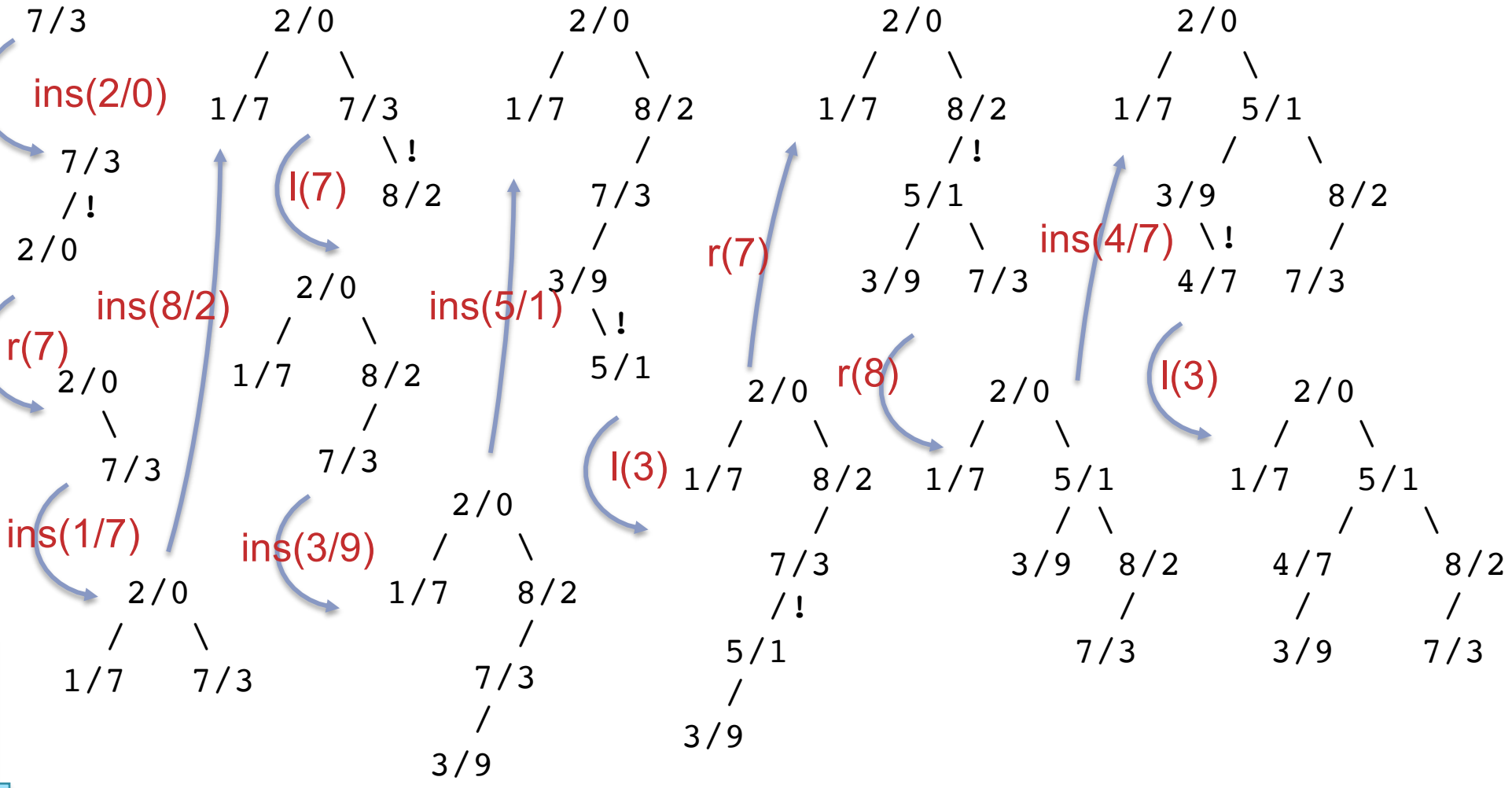


A treap is a binary search tree where the nodes have both user specified keys (k) and internally assigned priorities (p).

When inserting, use standard BST insertion algorithm, but then use rotations to iteratively move up the node while it has lower priority than its parent (analogous to a heap, but with rotations)

A (better) example

Insert the following pairs: 7/3, 2/0, 1/7, 8/2, 3/9, 5/1, 4/7



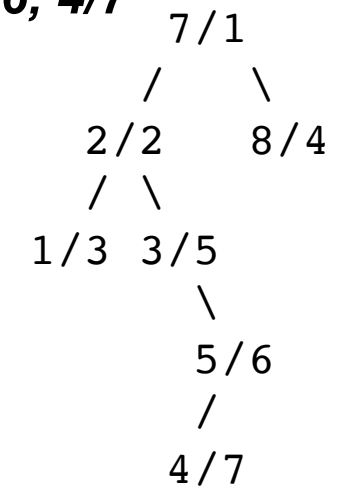
Notice that we inserted the same keys, but with different priorities

Just by changing the priorities, we can improve the balance!

Treap Reflections

Insert the following pairs: 7/1, 2/2, 1/3, 8/4, 3/5, 5/6, 4/7

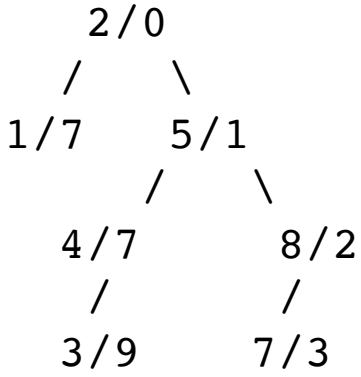
Since the priorities are in sorted order, becomes a standard BST and may have O(n) height



Insert the following pairs: 7/3, 2/0, 1/7, 8/2, 3/9, 5/1, 4/7

With a standard BST, for 2 to be root it would have to be the first key inserted, and 5 would have to proceed all the other keys except 1, …

It is as if we saw the sequence:
2,5,8,7,1,4,3
Note priorities in sorted order:
2/0, 5/1, 8/2/, 7/3, 1/7, 4/7, 3/9



What priorities should we assign to maintain a balanced tree?

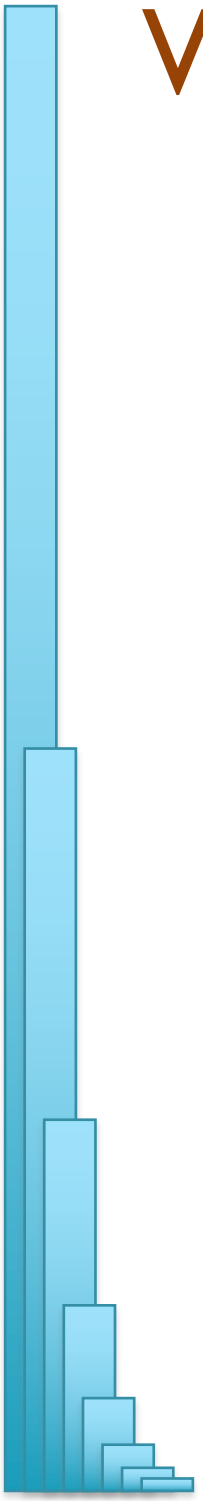
Math.random()

Using random priorities essentially shuffles the input data
(which might have bad linear height)

into a ***random permutation***
that we ***expect*** to have O(log n) height
☺

It is possible that we could randomly shuffle into a poor tree configuration, but that is extremely rare.

In most practical applications, a treap will perform just fine, and will often outperform an AVL tree that guarantees O(log n) height but has higher constants

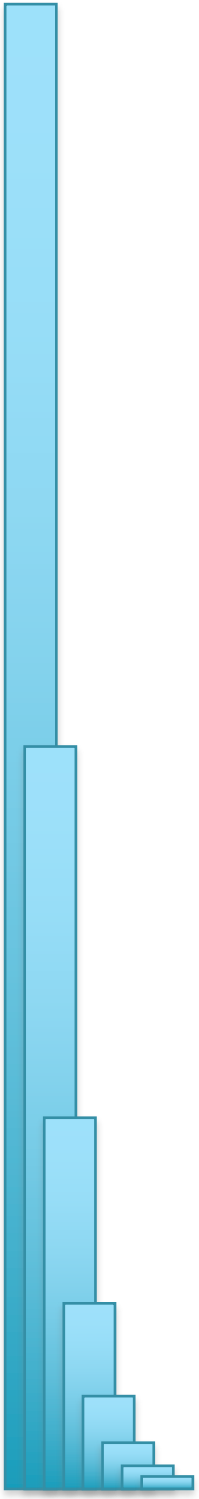


Self Balancing Trees

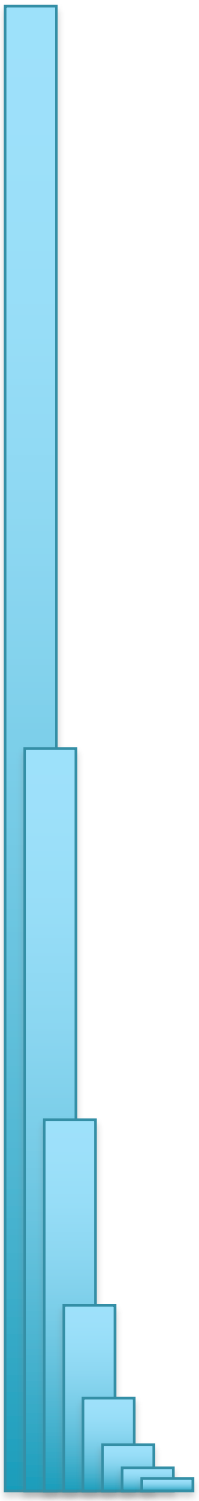
Understanding the distinction between different kinds of balanced search trees:

- ***AVL trees guarantee worst case O(log n)*** operations by carefully accounting for the tree height
- ***treaps guarantee expected O(log n)*** operations by selecting a random permutation of the input data
- ***splay trees guarantee amortized O(log n)*** operations by periodically applying a certain set of rotations (see lecture notes)

If you have to play it safe and don't trust your random numbers,
=> AVL trees are the way to go.
If you can live with the occasional O(n) op
=> splay trees are the way to go.
And if you trust your random numbers
=> treaps are the way to go.



Part 4: Hash Tables



Maps

aka dictionaries

aka associative arrays

```
Mike      -> Malone 323
Peter     -> Malone 223
Joanne     -> Malone 225
Zack      -> Malone 160 suite
Debbie     -> Malone 160 suite
Yair      -> 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

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…

Could we do everything in O(lg n) time or faster?
=> Balanced Search Trees

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>

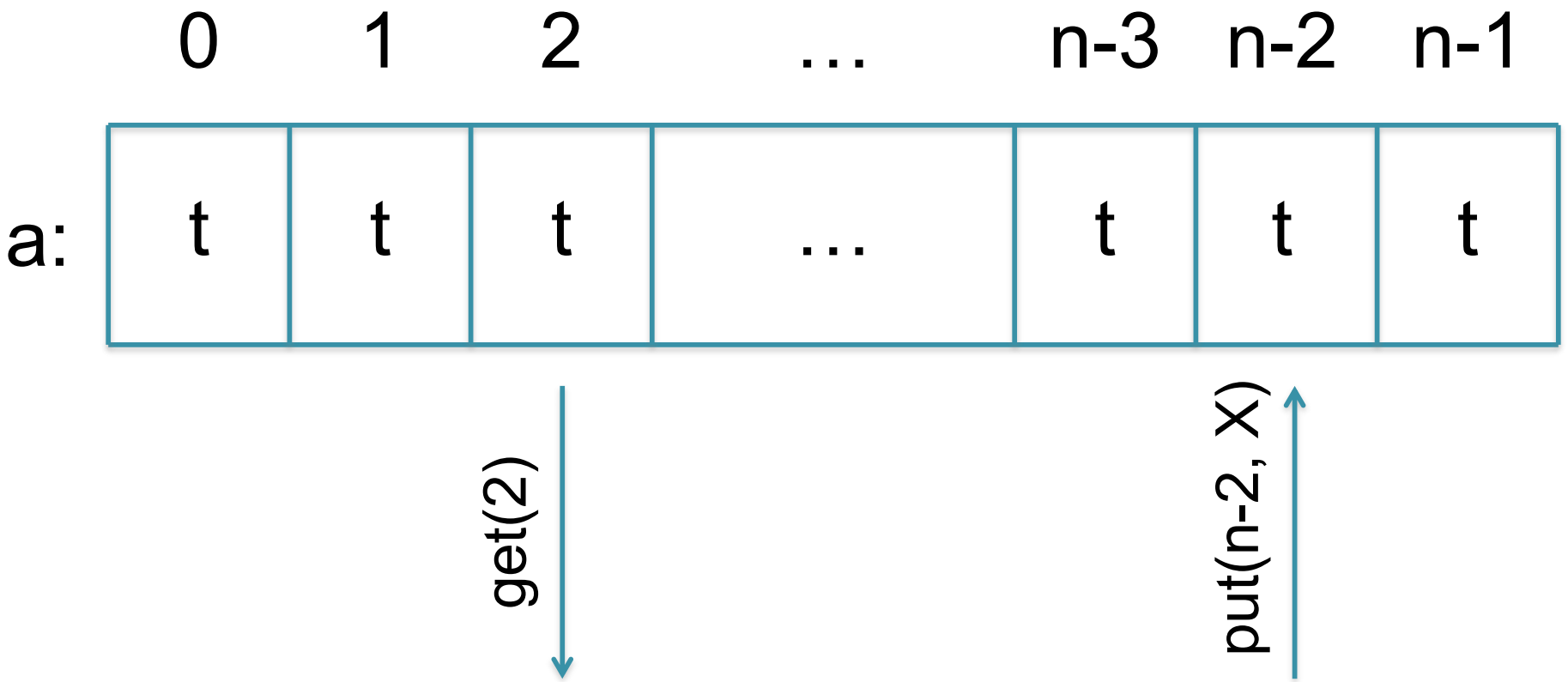
```
0   -> Mike
1   -> Peter
2   -> Joanne
3   -> Zack
4   -> Debbie
5   -> Yair
6   -> Ron
```

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

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

Could we do everything in O(lg n) time or faster?
=> Balanced Search Trees

ADT: Arrays



- Fixed length data structure
- Constant time get() and put() methods
- Definitely needs to be generic ☺

Hashing

Array[13] = 10; Array[42] = 15 ***O(1)***

Array["Mike"] = 10; Array["Peter"] = 15 ***BST:O(lg n) -> Hash:O(1)***

Hash Function enables Map<K, V> as Map<Integer, V> as Array<V>

- h(): K -> Integer for any possible key K
- h() should distribute the keys uniformly over all integers
- if k_1 and k_2 are "close", $h(k_1)$ and $h(k_2)$ should be "far" apart

Typically want to return a small integer, so that we can use it as an index into an array

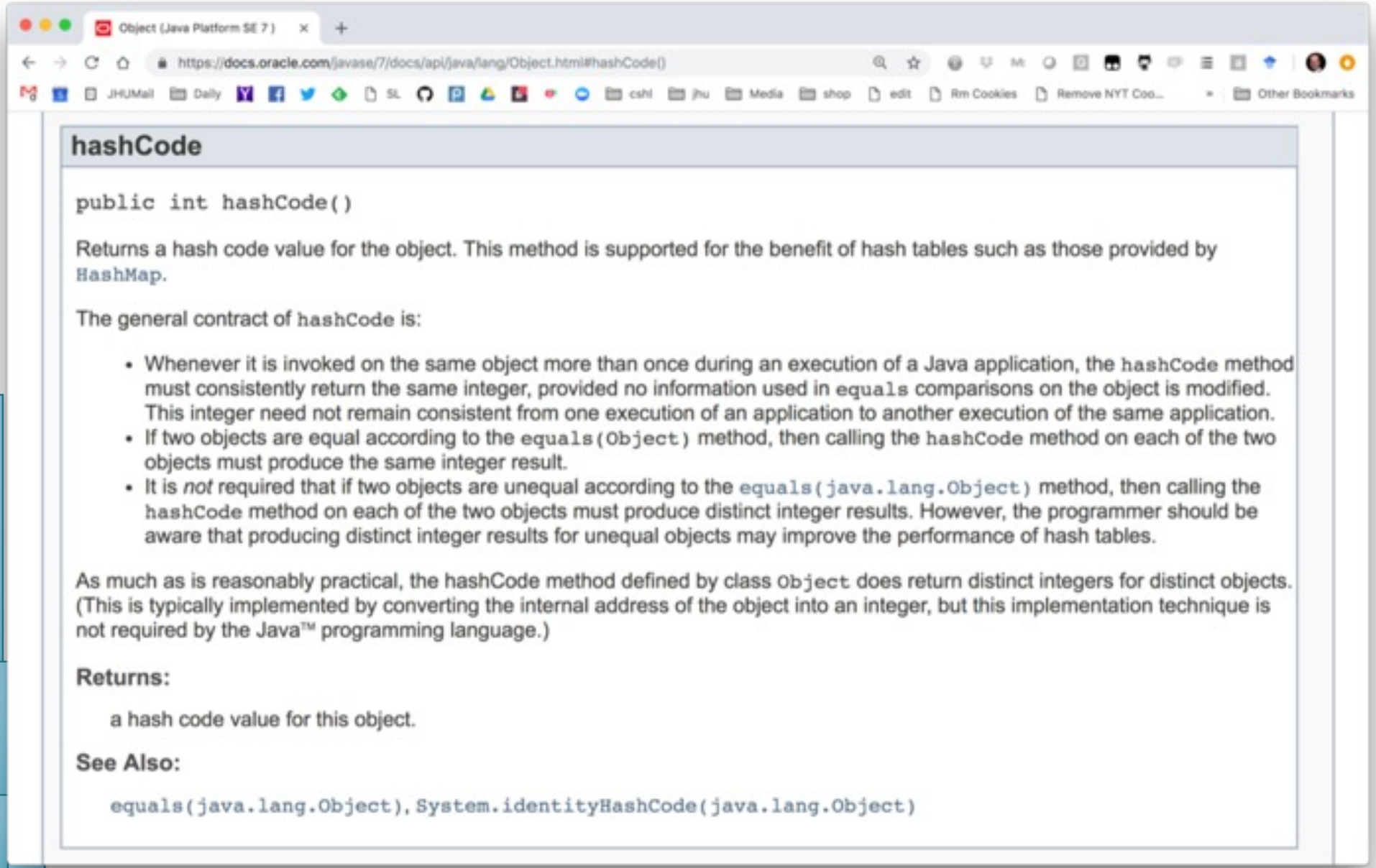
- An array with 4B cells in not very practical if we only expect a few thousand to a few million entries
- How do we restrict an arbitrary integer x into a value up to some maximum value n?

```
0 <= x % n < n
```

Compression function: c(i) = abs(i) % length(a)

Transforms from a large range of integers to a small range (to store in array a)

Where do we get keys from?



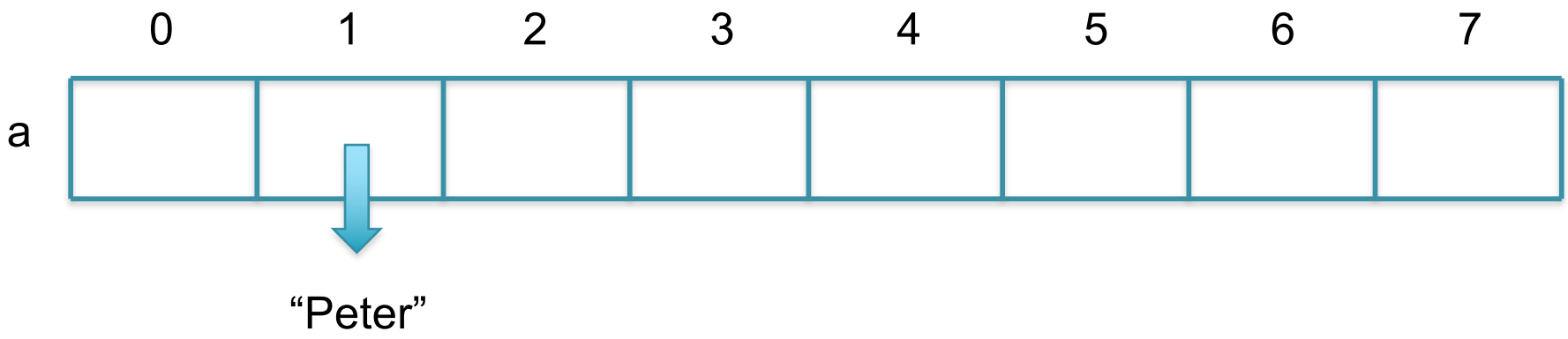
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



insert(1, "Peter"): c(h(1)) = c(1) = 1

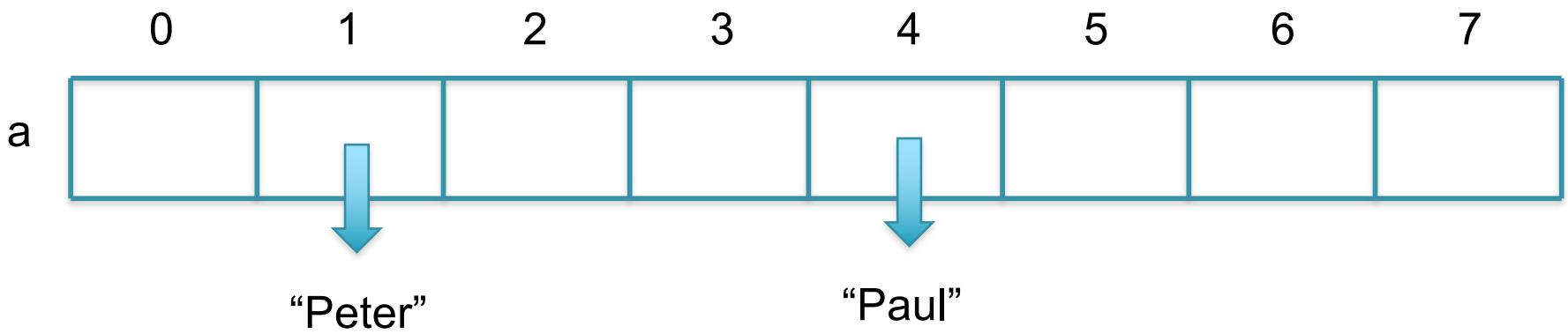
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



Insert(20, "Paul"): c(h(20)) = c(20) = 4

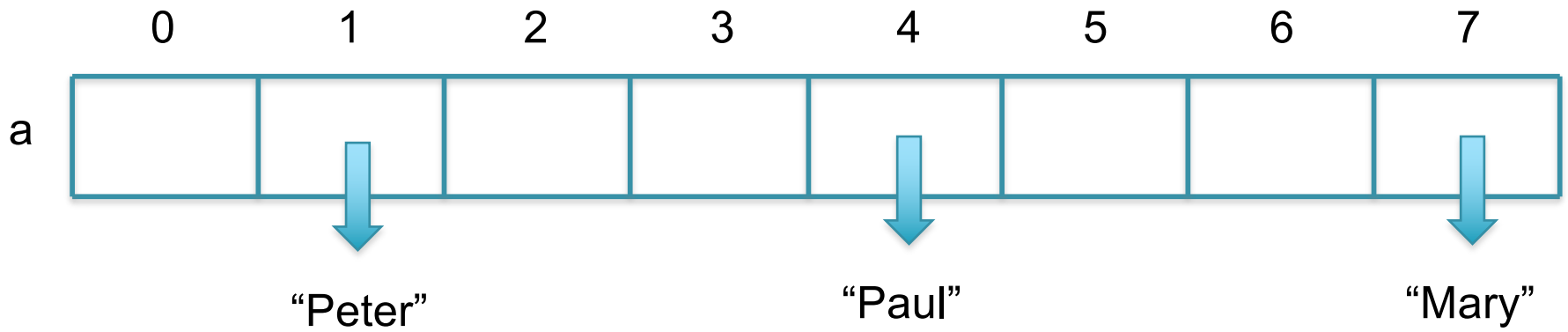
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



insert(15, “Mary”): c(h(15)) = c(15) = 7

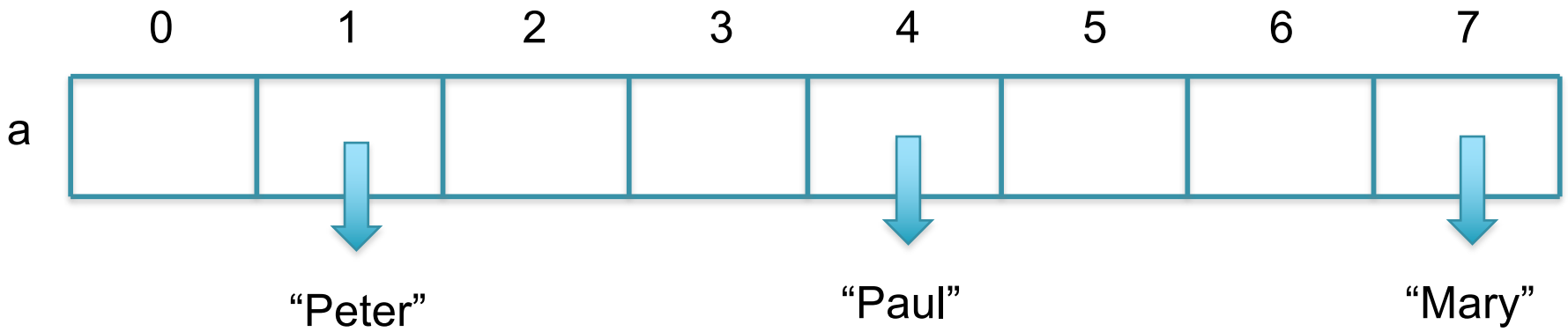
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



get(15): get(c(h(15))) = get(c(15)) = get(7) => "Mary" ☺

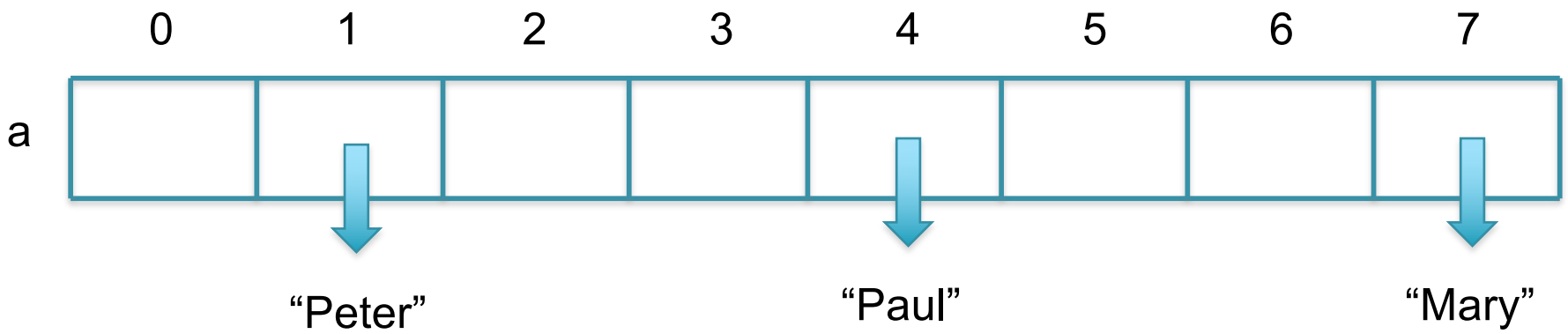
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



has(4): has(c(h(4)) = has(c(4)) = has(4) => "Paul" ☹

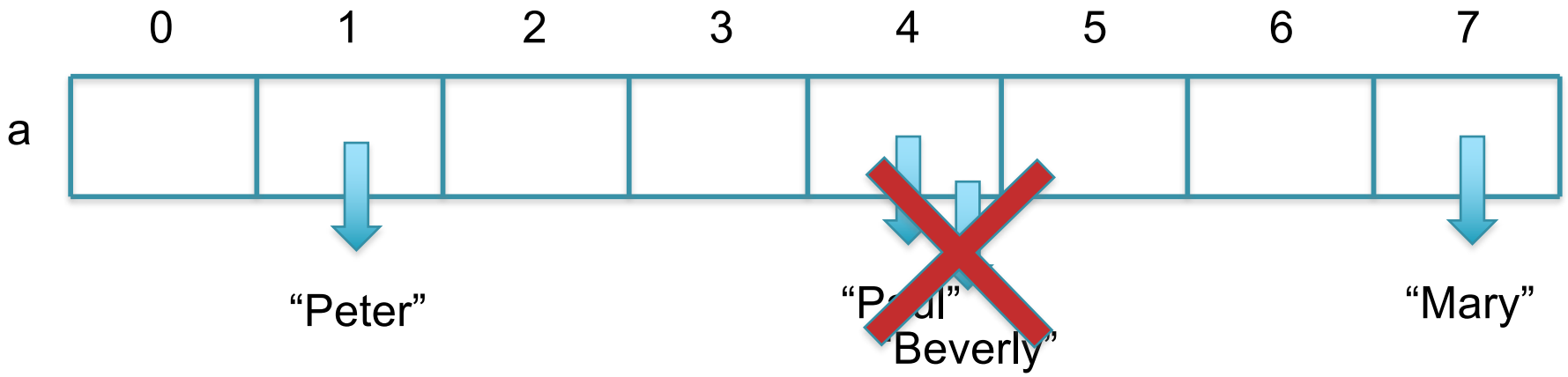
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



insert(4, "Beverly"): c(h(4)) = c(4) = 4 ☹

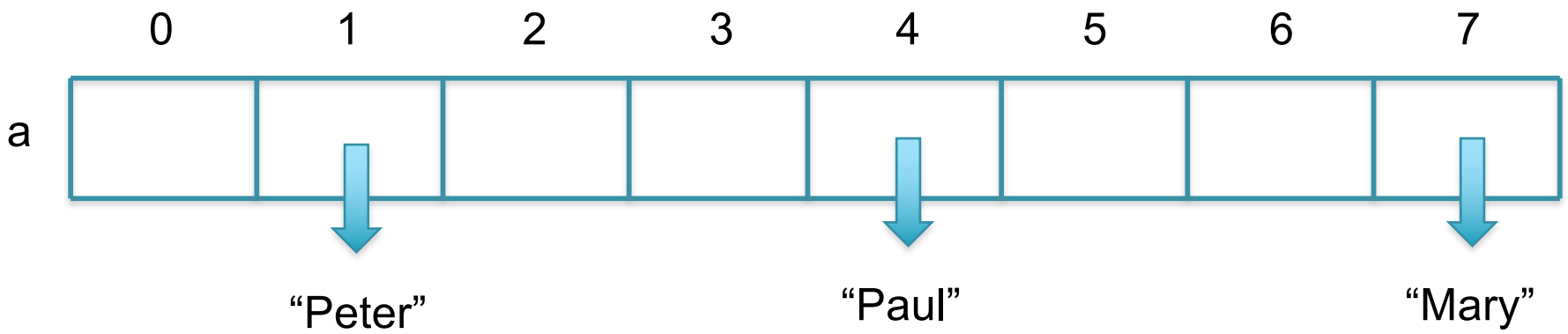
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



Two problems caused by collisions:

False positives: How do we know the key is the one we want?

Collision Resolution: What do we do when 2 keys map to same location?

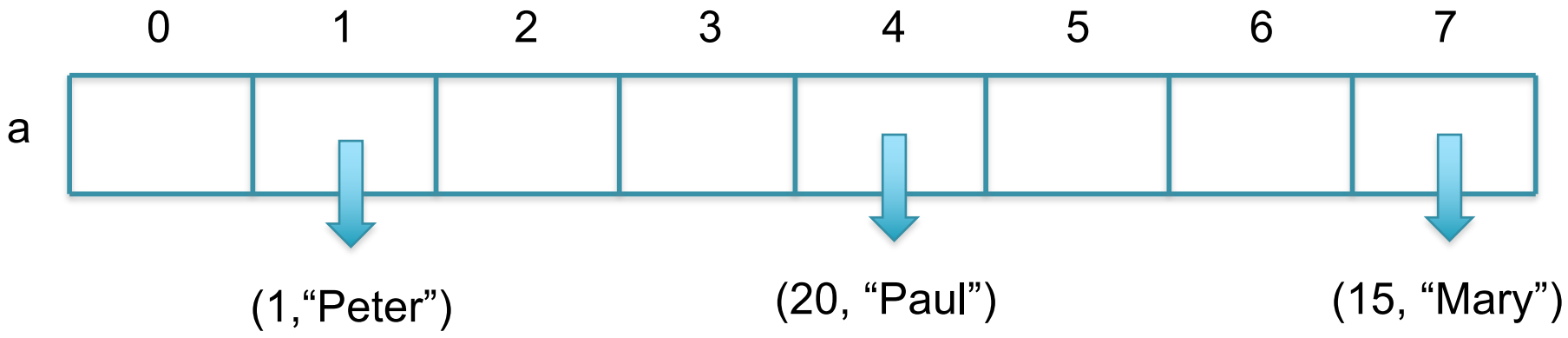
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



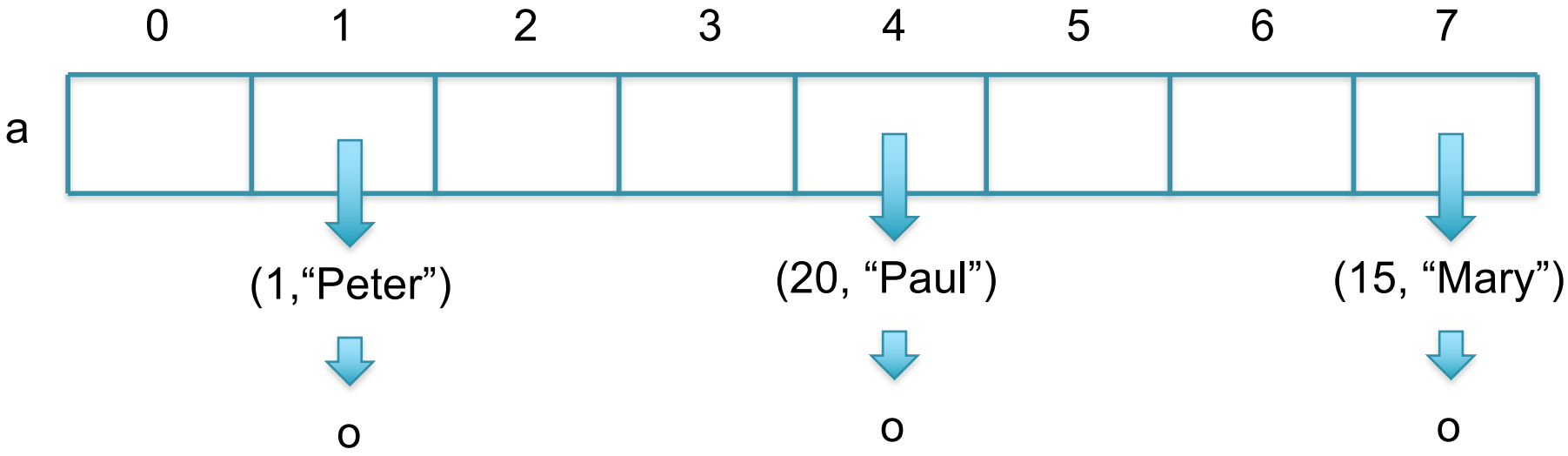
Two problems caused by collisions:

False positives: How do we know the key is the one we want?

Collision Resolution: What do we do when 2 keys map to same location?

Separate chaining

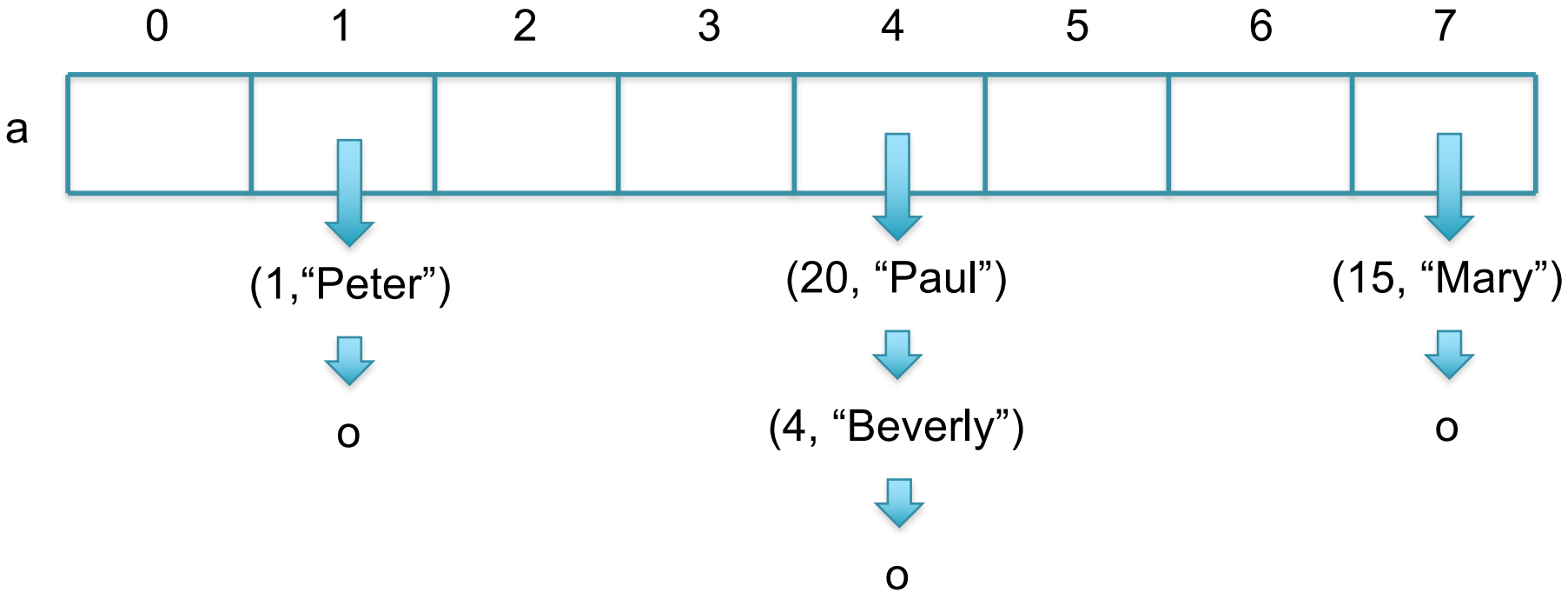
Use Array<List<V>> instead of an Array<V> to store the entries



insert(4, “Beverly”): c(h(4)) = c(4) = 4

Separate chaining

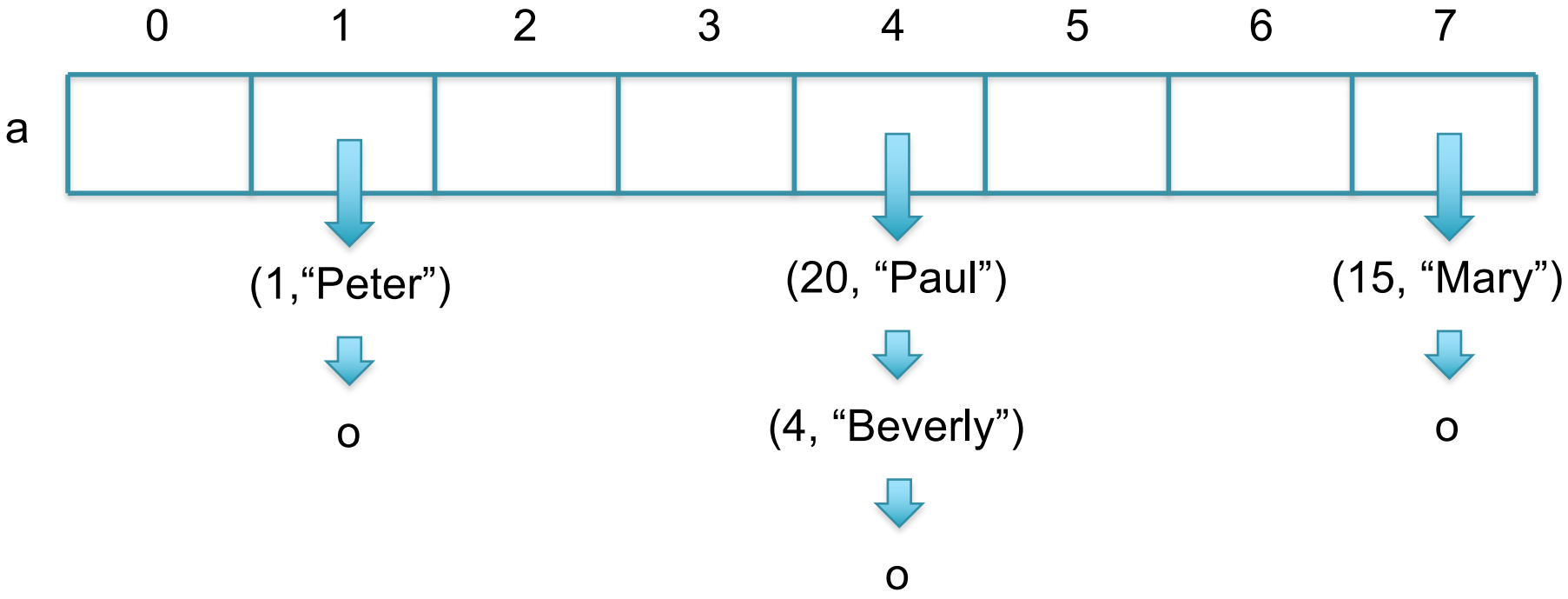
Use Array<List<V>> instead of an Array<V> to store the entries



insert(4, “Beverly”): c(h(4)) = c(4) = 4

Separate chaining

Use Array<List<V>> instead of an Array<V> to store the entries



Using separate chaining we could implement the Map<K,V> interface ☺

Seems fast, but how fast do we expect it to be?

Separate Chaining Analysis

Assume the table has just 1 cell:

All n items will be in a linked list => O(n) insert/find/remove ☹

Assume the hash function h() always returns a constant value

All n items will be in a linked list => O(n) insert/find/remove ☹

Assume table has m cells AND h() evenly distributes the keys

- Every cell is equally likely to be selected to store key k, so the n items should be evenly distributed across m slots
- Average number of items per slot: n/m
  - Also called the ***load factor*** *(*commonly written as α)
  - Also the probability of a collision when inserting a new key
    - Empty table: 0/m probability
    - After 1st item: 1/m
    - After 2nd item: 2/m

Expected time for unsuccessful search:	O(1+n/m)
Expected time for successful search:	O(1+n/m/2) => O(1+n/m)

If n < c*m, then we can expect constant time! ☺

Next Steps

1. Work on HW7

2. Check on Piazza for tips & corrections!