The Hash Array-Mapped Trie (HAMT)

Kris Micinski

Logistics

- I’m (probably) gone next Tuesday
- This Thursday: course / exam review
- Next Thursday (probably): present projects / competition
- None of the HAMT particulars will be on the exam

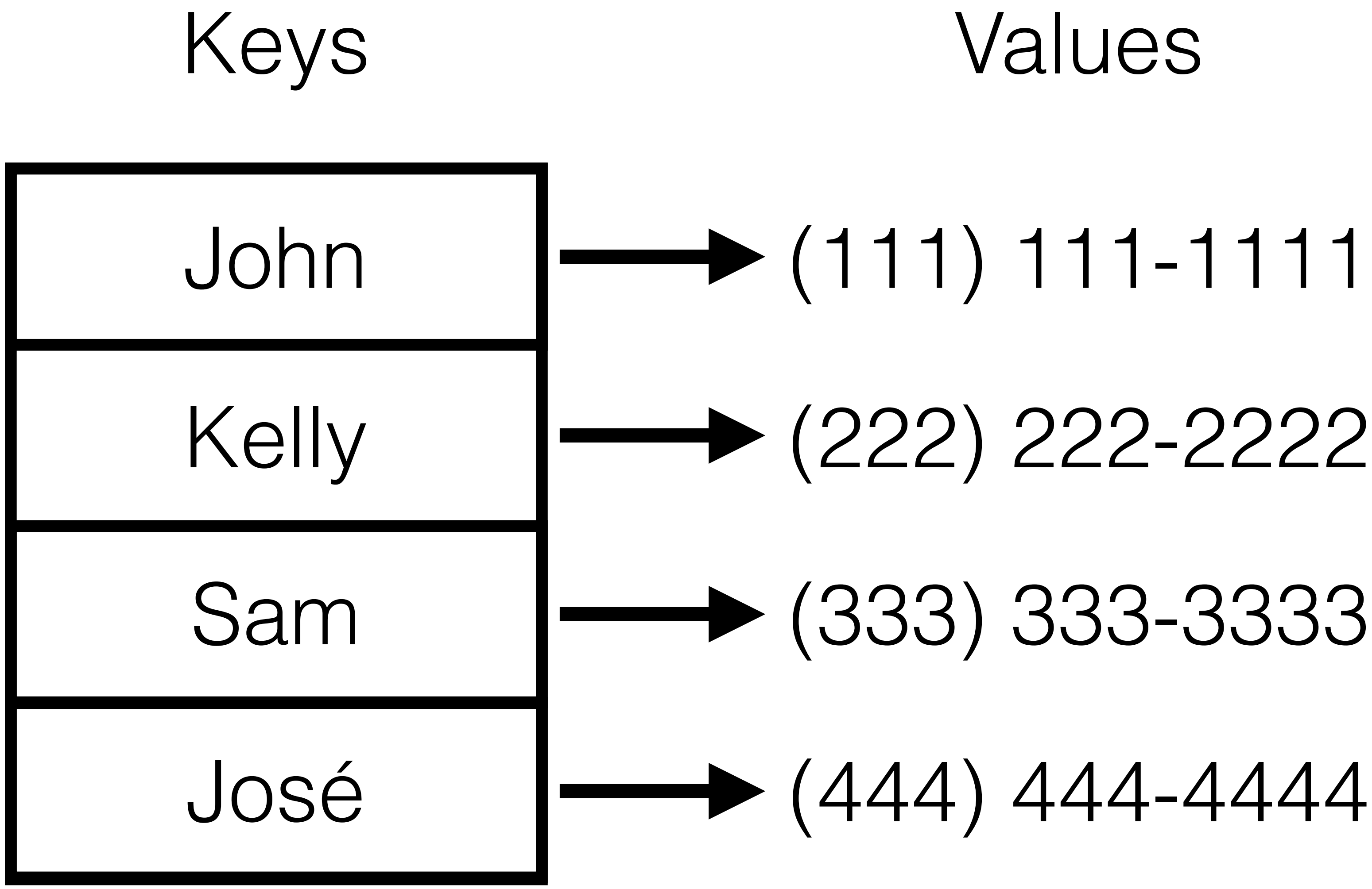
HAMT — High-Level Benefit

- **Persistent** hash map / set with:
  - Constant time insertion
  - Constant time lookup
  - Robust cache performance

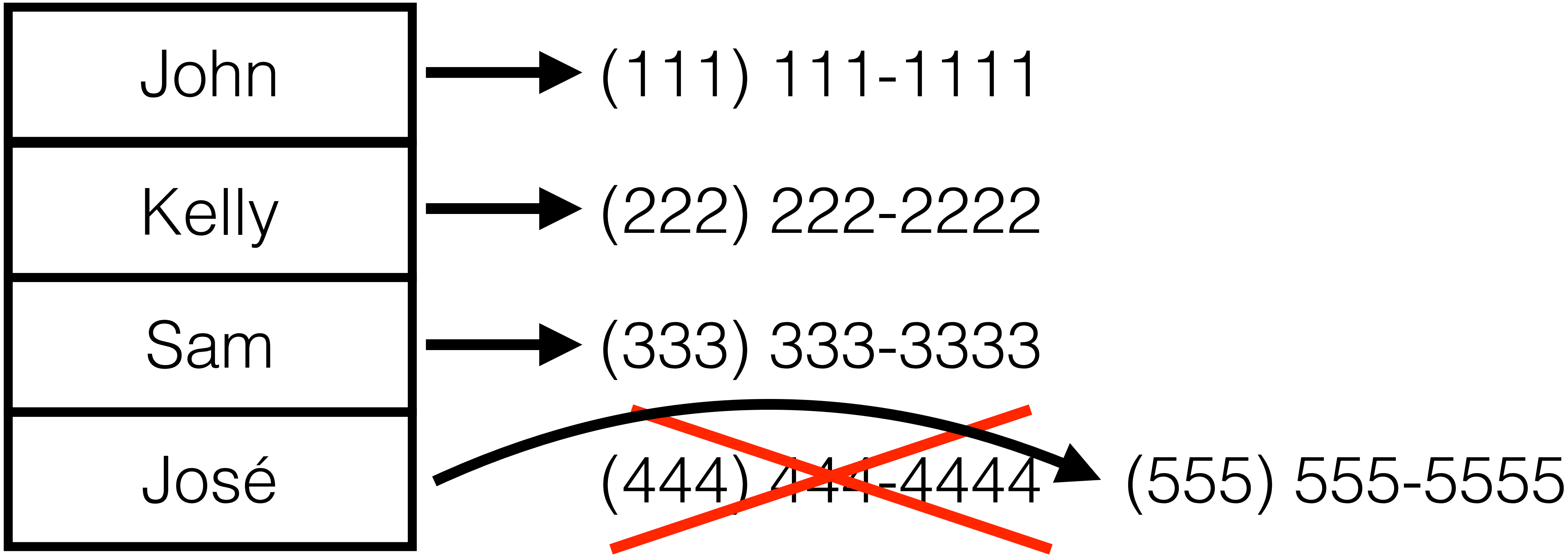
Problems with other DS

- Lookup / insertion for **balancing** binary trees:
  - log(n) with **imperative** version
  - Not persistent data structure
- Same thing with hash tables…
- Coming up with persistent hash map is hard!

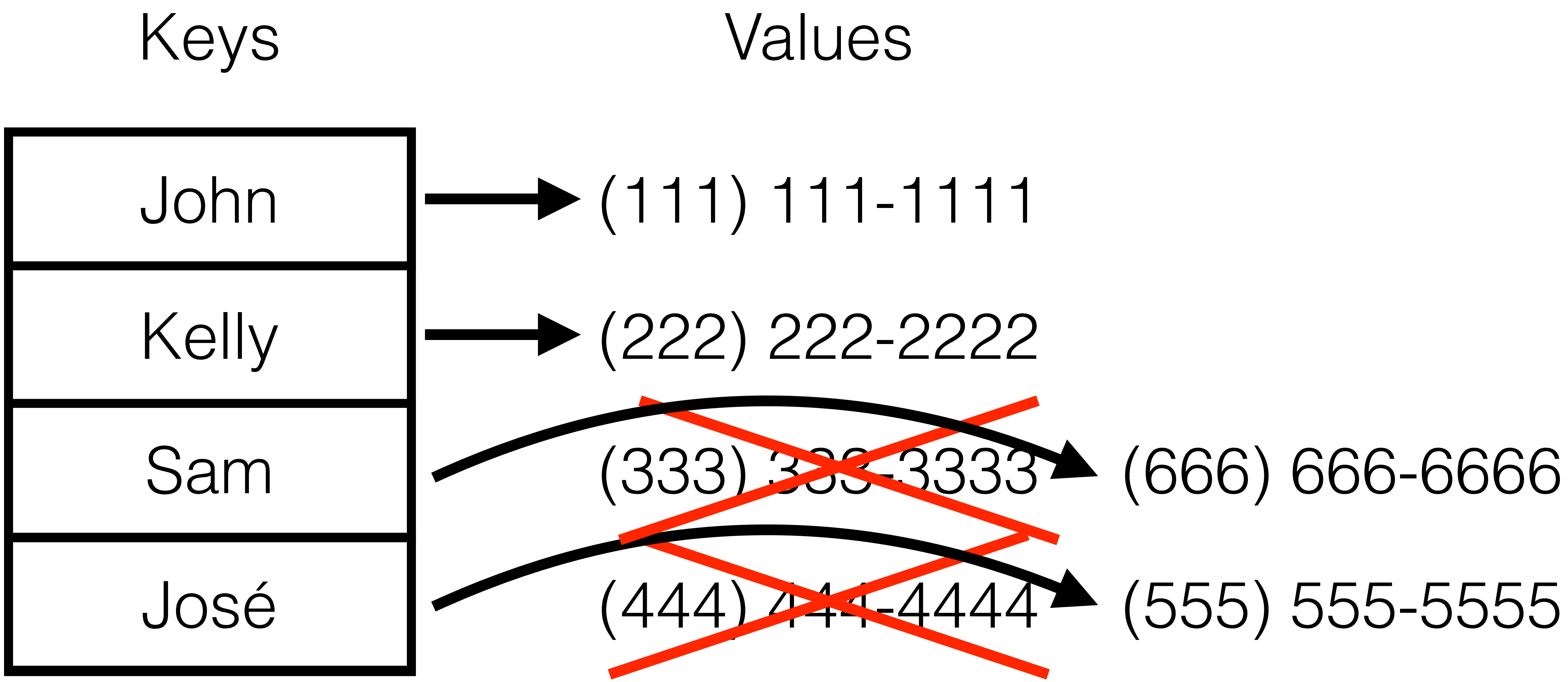
Motivation: phone books over time…



Keys	Values
John	(111) 111-1111
Kelly	(222) 222-2222
Sam	(333) 333-3333
José	(444) 444-4444



Keys	Values
John	(111) 111-1111
Kelly	(222) 222-2222
Sam	(333) 333-3333
José	~~(444) 444-4444~~ (555) 555-5555



Keys

Values

John

Kelly

Sam

José

(111) 111-1111

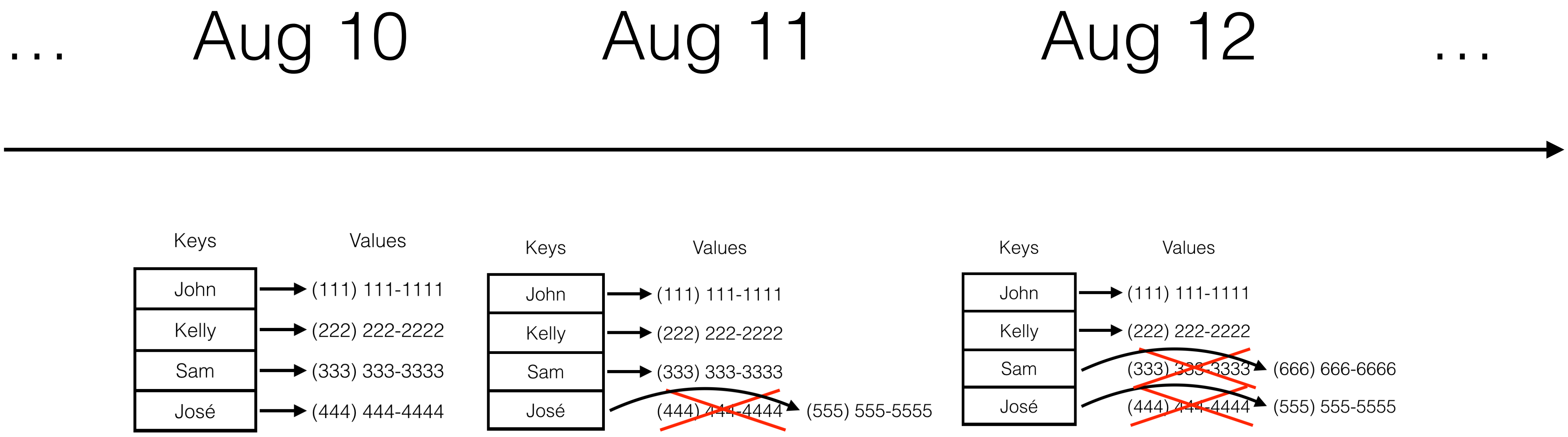
(222) 222-2222

~~(333) 333-3333~~

~~(444) 444-4444~~

(666) 666-6666

(555) 555-5555



… Aug 10 Aug 11 Aug 12 …

Aug 10

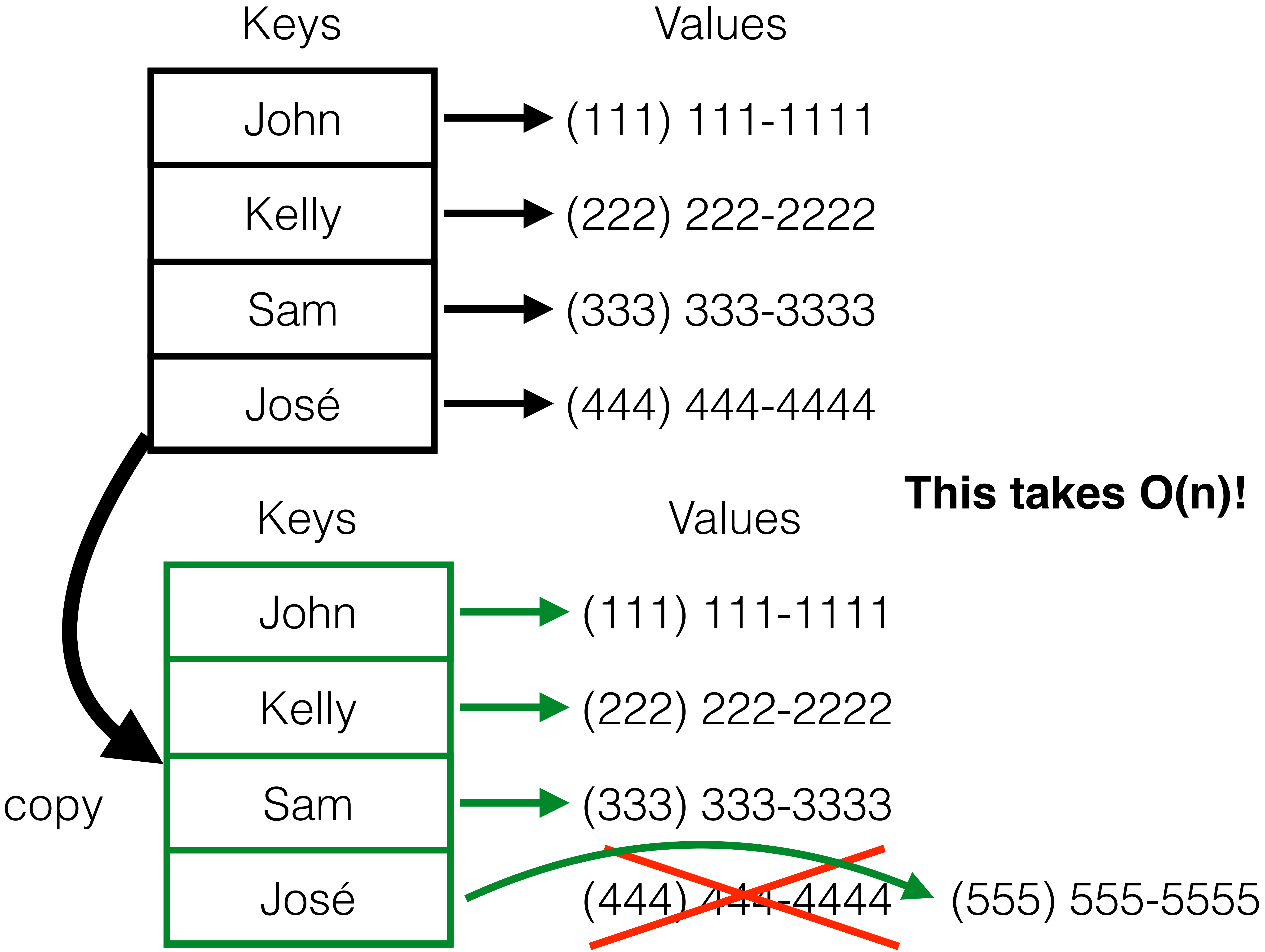
Keys	Values
John	(111) 111-1111
Kelly	(222) 222-2222
Sam	(333) 333-3333
José	(444) 444-4444

Aug 11

Keys	Values
John	(111) 111-1111
Kelly	(222) 222-2222
Sam	(333) 333-3333
José	~~(444) 444-4444~~ (555) 555-5555

Aug 12

Keys	Values
John	(111) 111-1111
Kelly	(222) 222-2222
Sam	~~(333) 333-3333~~ (666) 666-6666
José	~~(444) 444-4444~~ (555) 555-5555



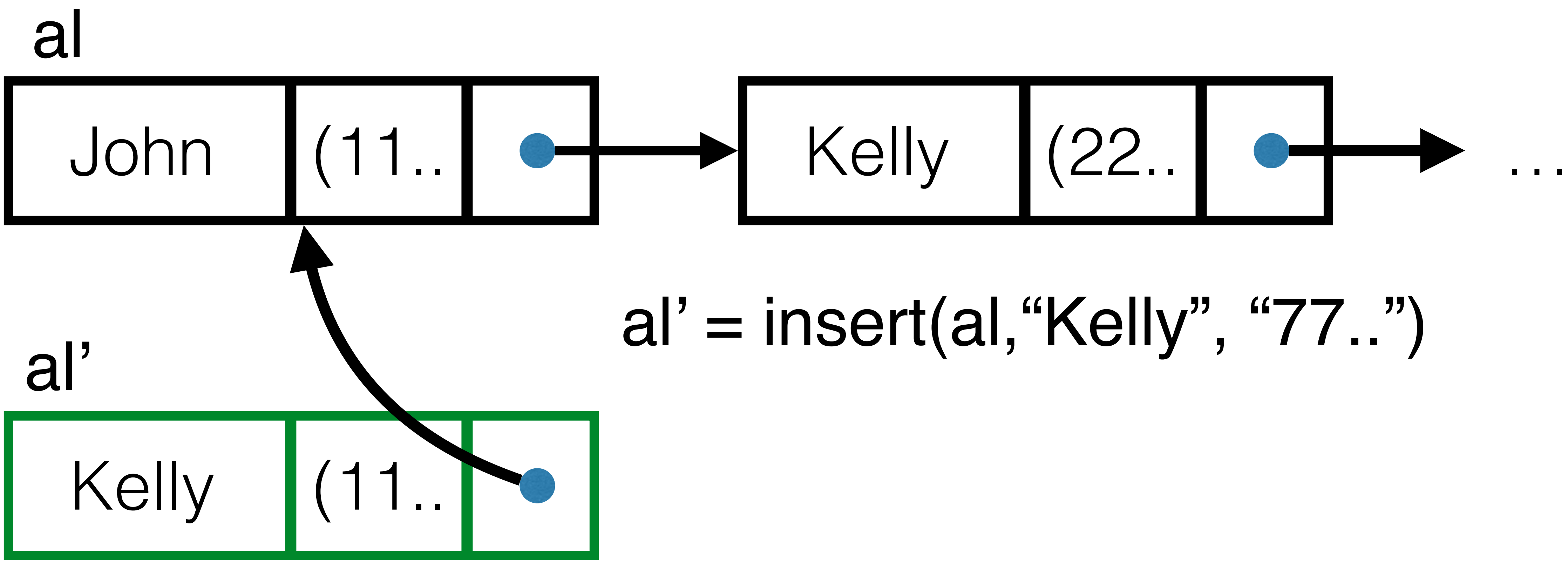
Keys	Values
John	(111) 111-1111
Kelly	(222) 222-2222
Sam	(333) 333-3333
José	(444) 444-4444

copy

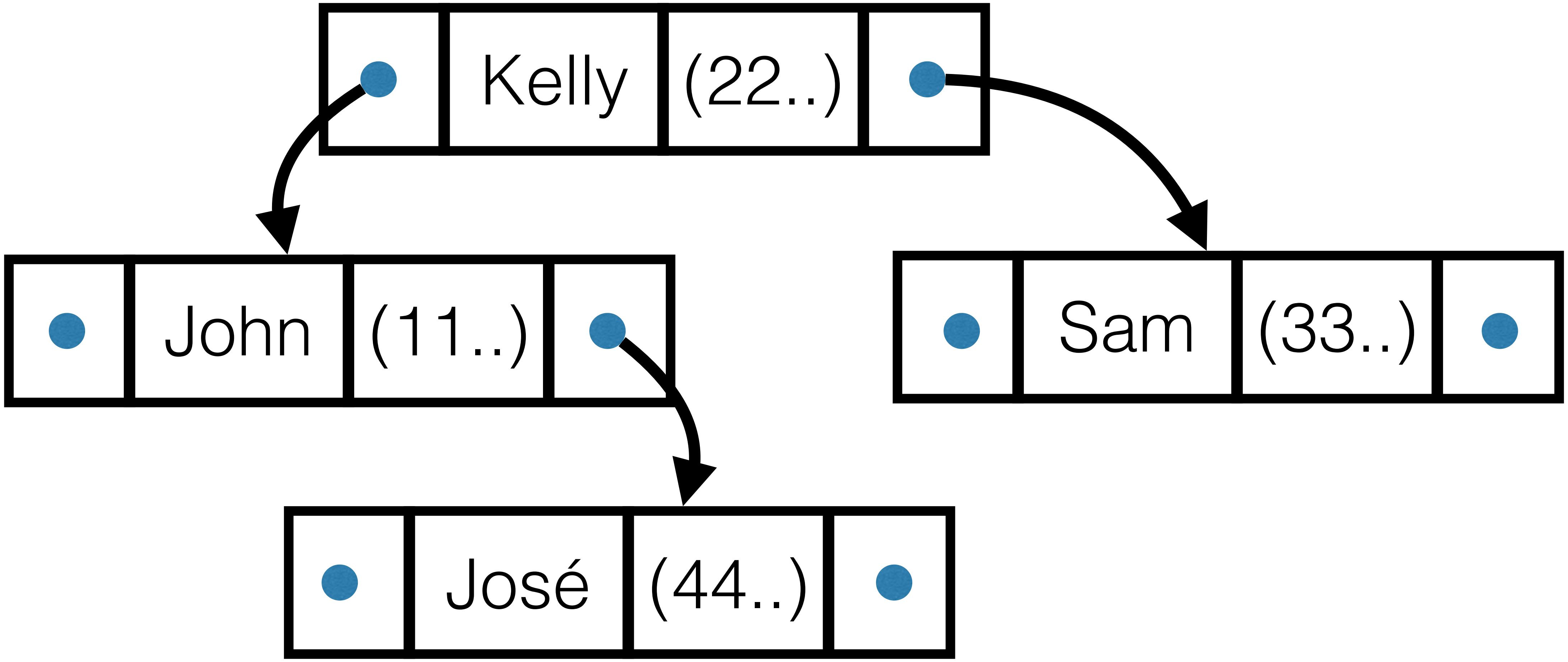
This takes O(n)!

Keys	Values
John	(111) 111-1111
Kelly	(222) 222-2222
Sam	(333) 333-3333
José	~~(444) 444-4444~~ (555) 555-5555

Inserting into association list: either O(1) or O(n)
Depends if you want time or space…

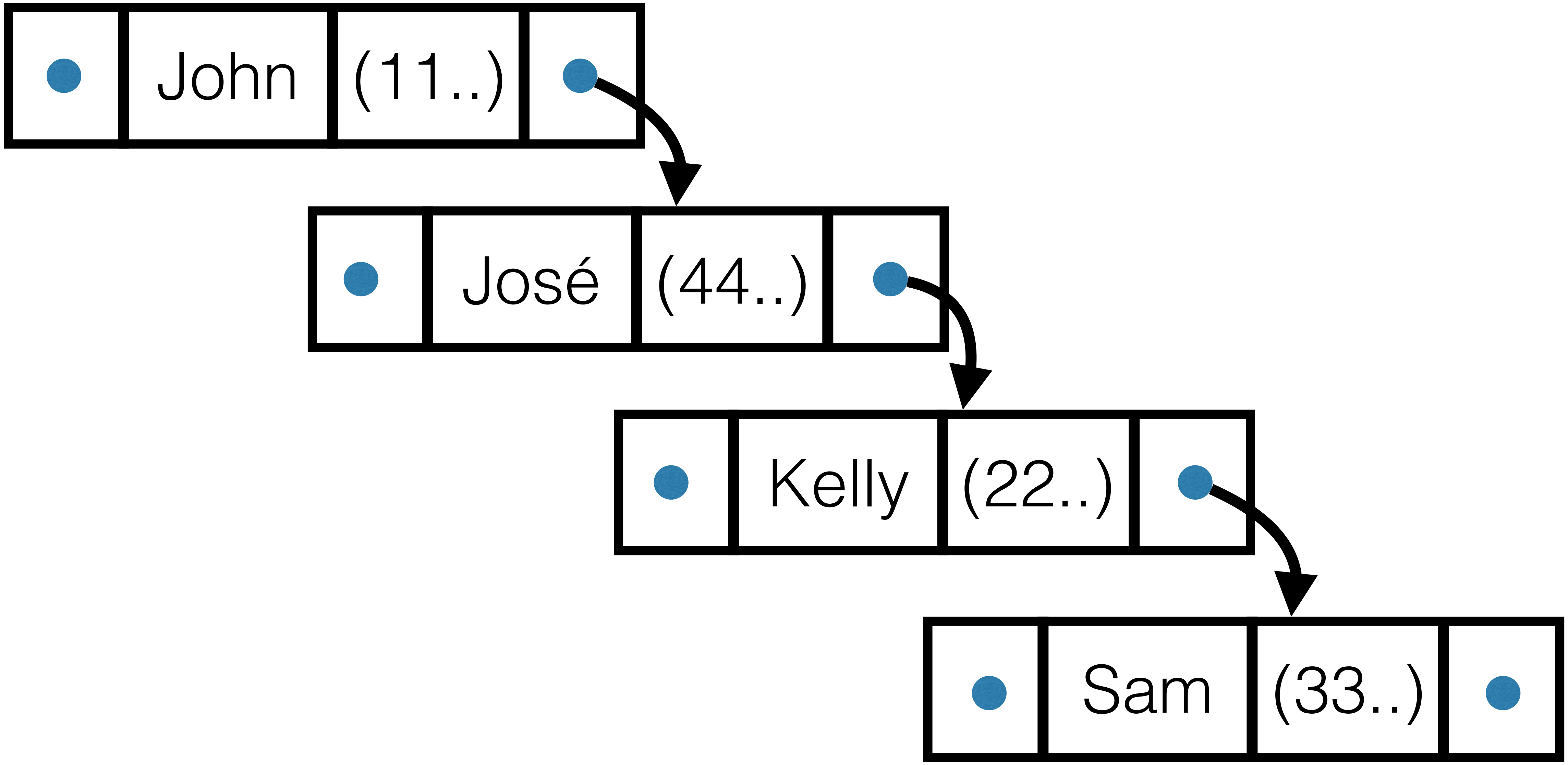


al’ = insert(al,“Kelly”, “77..”)



Balanced binary trees are good….

But unbalanced binary trees are not…!



(Naive insertion **potentially** leads to O(n)!)

What’s one way to **approximate** balanced binary trees?

(I.e., if I want to insert n names and end up with an “almost” balanced tree?)

Insight: **randomize** insertion order

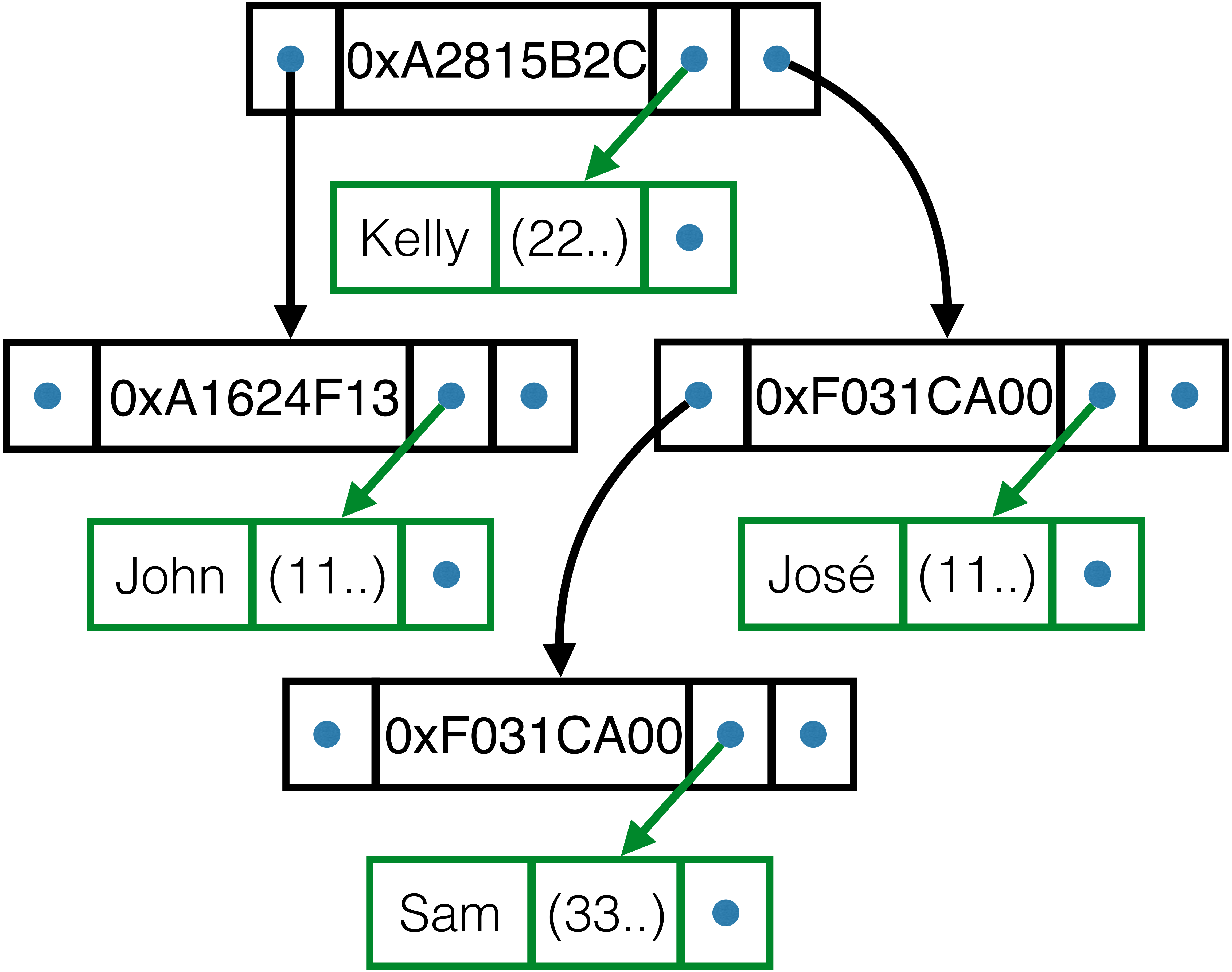
I can do this by storing a **hash**!

Now: each node of our binary tree stores a **hash** and an **association list**

(Why do you need this?)

Observation 1

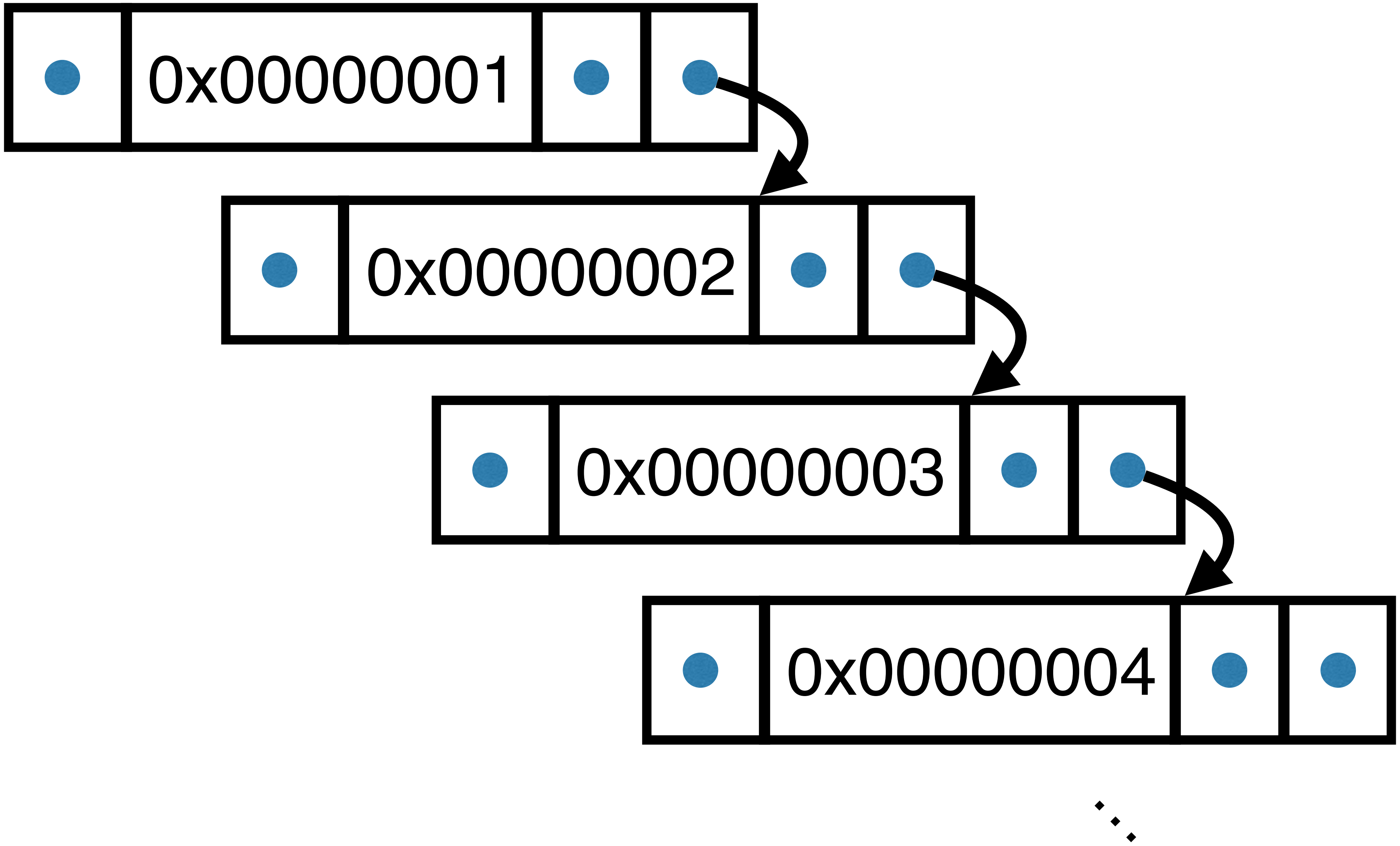
Store hashes as keys into a tree, back it with association list

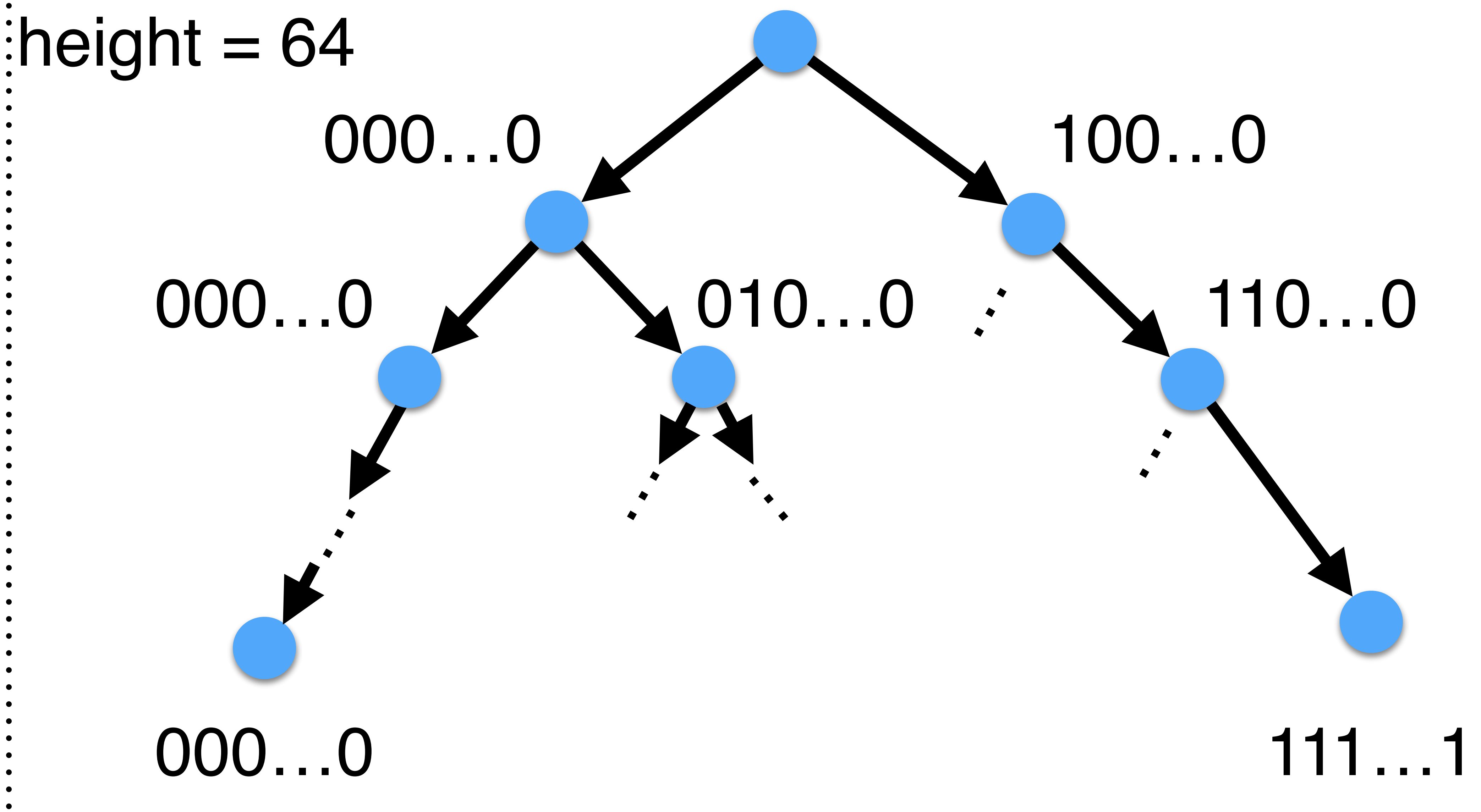

0xA2815B2C
Kelly
(22..)
0xA1624F13
0xF031CA00
John
(11..)
José
(11..)
0xF031CA00
Sam
(33..)

Observation 2

Instead of tree, why not use a **trie?**

0x10000001
↘
0x10000002
↘
0x10000003
↘
0x10000004
↘
0x10000005
↘
…


0x00000001
0x00000002
0x00000003
0x00000004
⋱



Now each successive bit in the trie indexes a successive bit in the hash

(Of course, each leaf still needs to be backed by assn. list!)

Observation 2

Now lookup takes **at most** 64!
(Because that’s how long our hash is!)

Ergo: Insertion takes ~O(1) time!

Observation 3

64 is still pretty crummy constant factors!

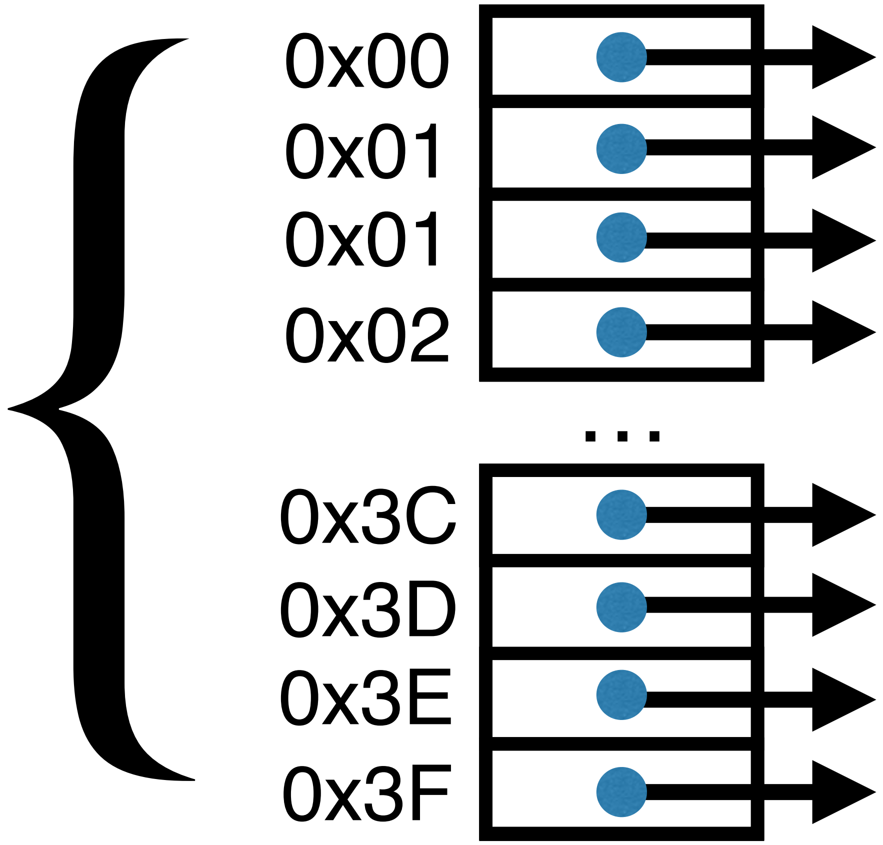
And it **always** takes 64 if we use a trie!

How can we do better..?

How can we do better..?

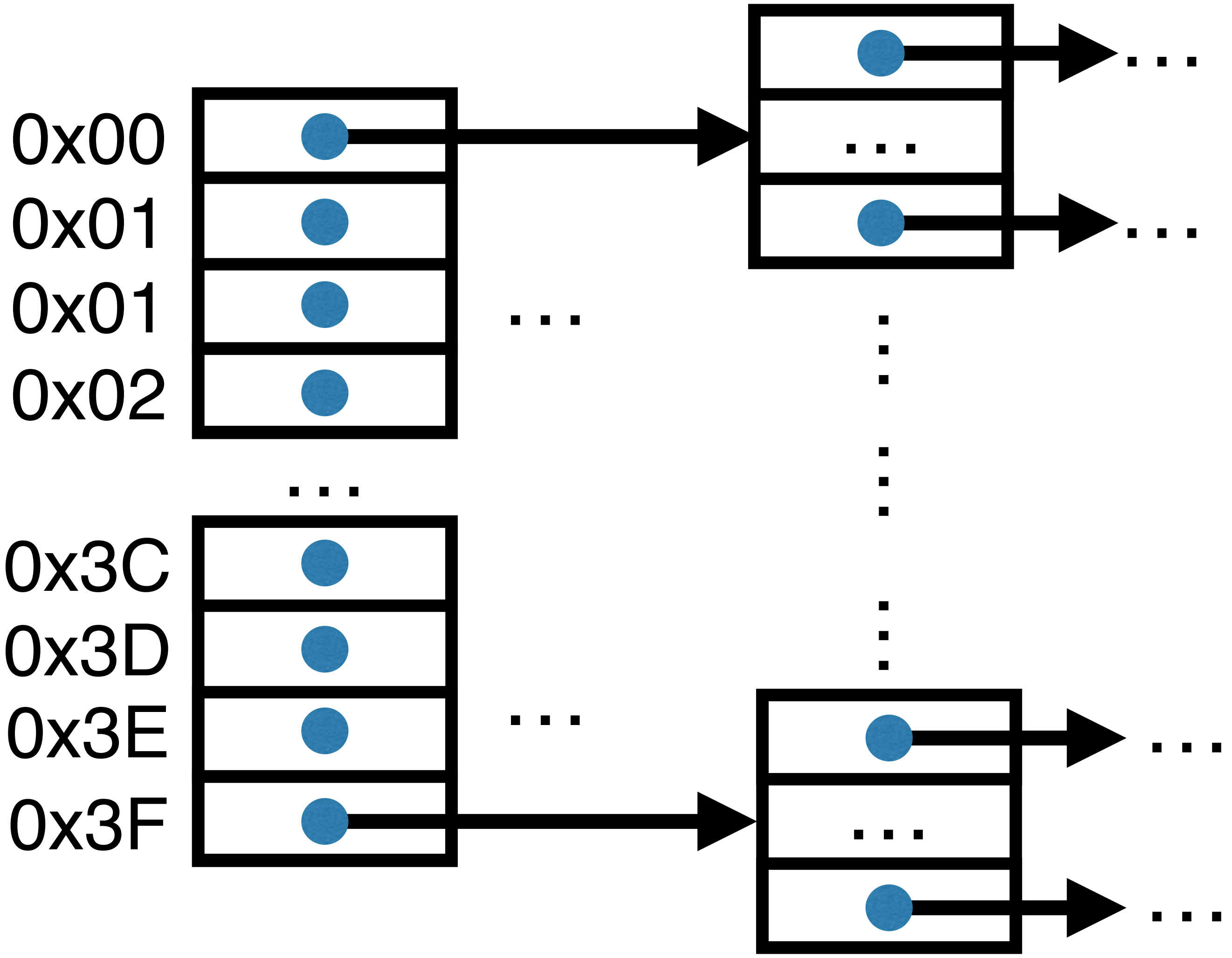
Idea: store more than 1 bit per node!

Store 64 subtries per node



(Can vary this up / down)

Now our trie looks like this!



To insert, walk over **chunks** of the hash!

(Walk through on board…)

Observation 3

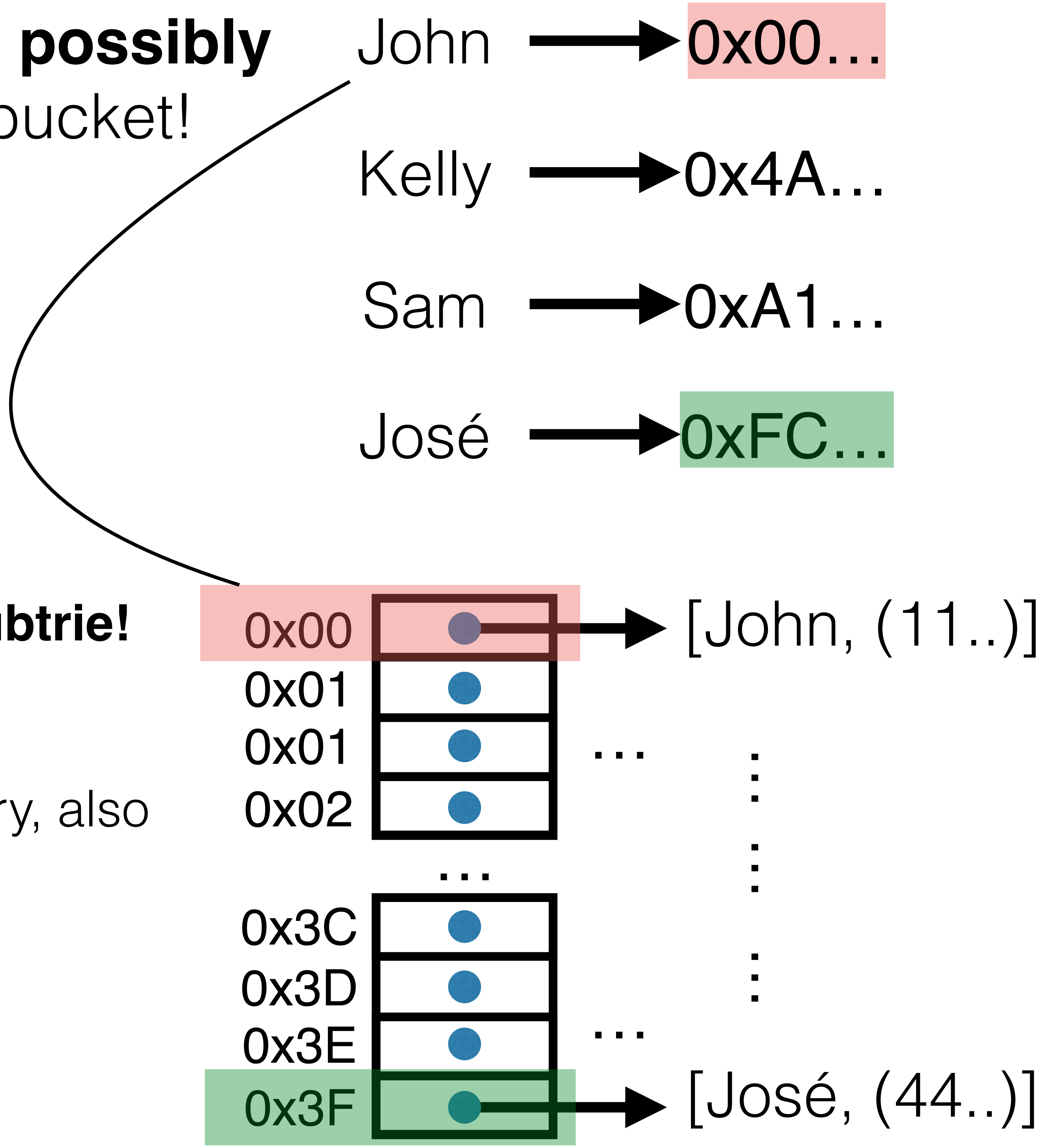
Get better constant factors by storing more subtries per node

(Downside: each node takes more memory)

Observation 4

Don’t store the **whole** trie until we really **need** to

No other keys could **possibly** be stored in this bucket!



John → 0x00…

Kelly → 0x4A…

Sam → 0xA1…

José → 0xFC…

0x00 → [John, (11..)]

0x01

0x01

0x02

…

0x3C

0x3D

0x3E

0x3F → [José, (44..)]

So don't store a subtrie!

Takes up less memory, also faster!

Question now: how do we implement insert?

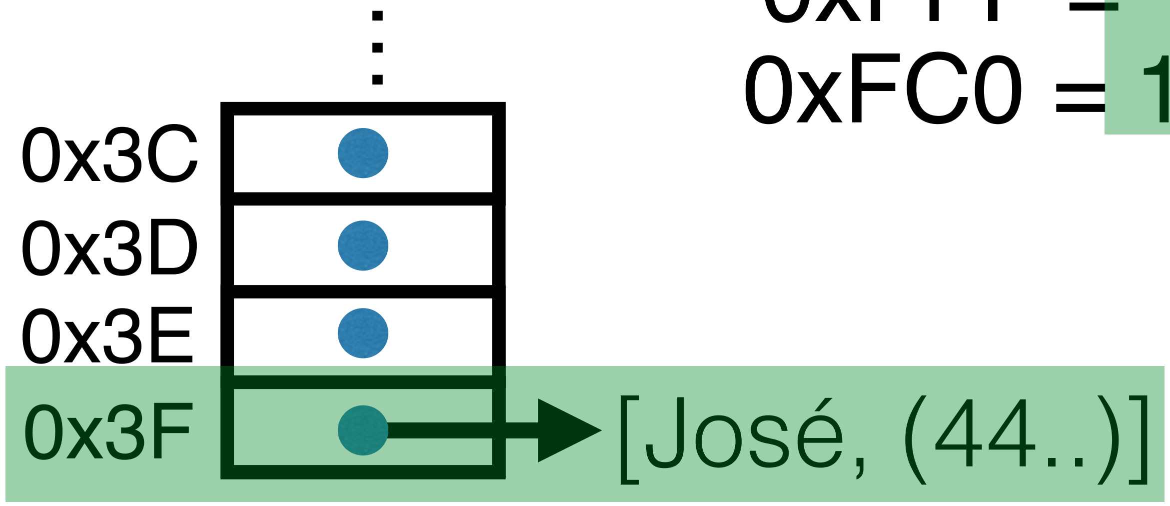
Observation: form **new** subtries in case of collision at some node insert(map,”Sam”, “(33..”)

Sam → 0xFFF…

José → 0xFC0…

0xFFF = 111111 111111

0xFC0 = 111111 000000



Split!

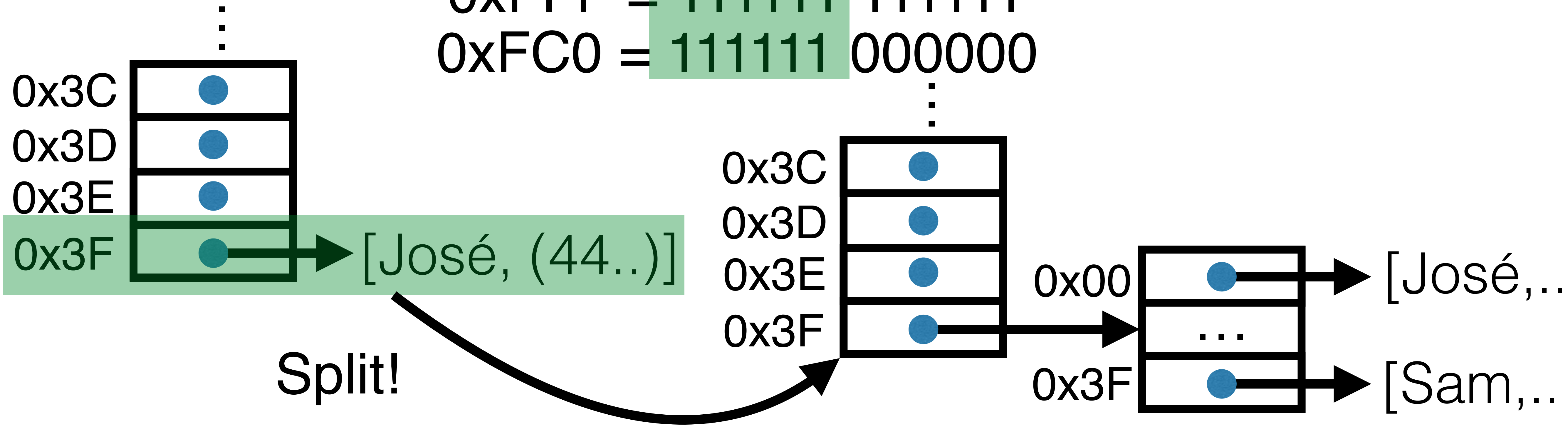
insert(map,”Sam”, “(33..”)

Sam → 0xFFF…

José → 0xFC0…

0xFFF = 111111 111111

0xFC0 = 111111 000000



0x3C

0x3D

0x3E

0x3F

[José, (44..)]

Split!

0x3C

0x3D

0x3E

0x3F

0x00

…

0x3F

[José,..

[Sam,..

Observation 4

Don’t store the **whole** trie until we really **need** to

(Reduces space the trie takes up! Storing fewer keys = fewer places taken up!)

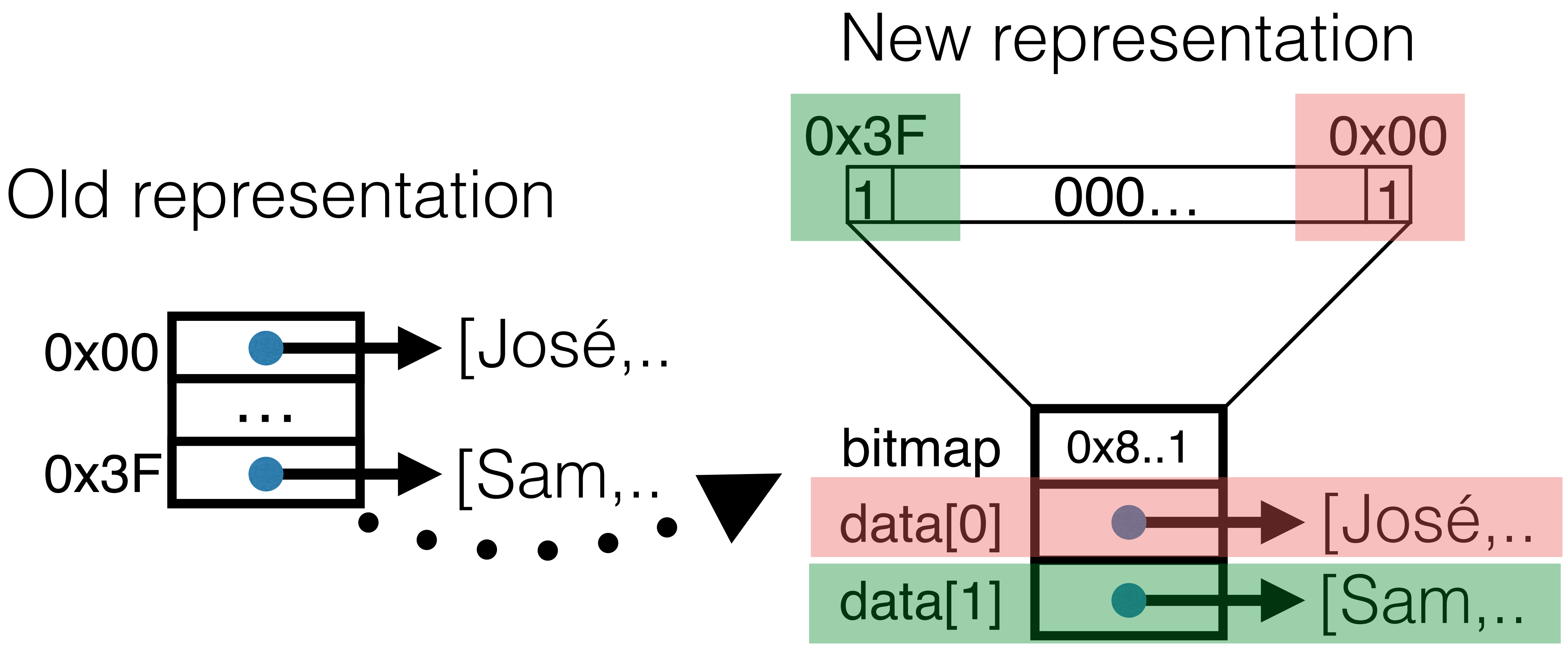
Observation 5

Storing 64 subtries is **still** very expensive!

Solution: store array of subtries, length of array is n where n is number of occupied buckets!

(Kris illustrates on board…)

Idea: use **bitmap!**



Putting it all together

Store hashes as keys into a tree, back it with association list

Instead of tree, use trie

Lower constant factors: store n (= 64 in our ex) subtries
(Con: more space for each subtrie!)

Two clever hacks:

- Don’t store subtries until you **absolutely need** to
- Use bitmap to reduce the need for n subtries until needed

Result

- Fast persistent hash map!
- Great constant factors
- Low overhead when it’s not storing much stufff
- Higher overhead as it fills up, but never far past constant factor!
- Useful in implementing interpreters, any other place when you need efficient hash map
- Most of the time a regular HT is probably fine, but think of HAMT!