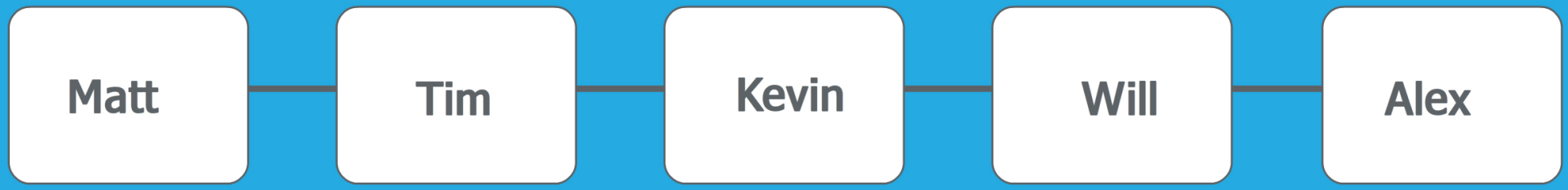
redis

a
data structure
server

Rob Tandy

Lunch and Learn
2012-10-16

linked list



element access: scales with # elements $O(n)$

push, pop: fast. independant of **n** $O(1)$

height:

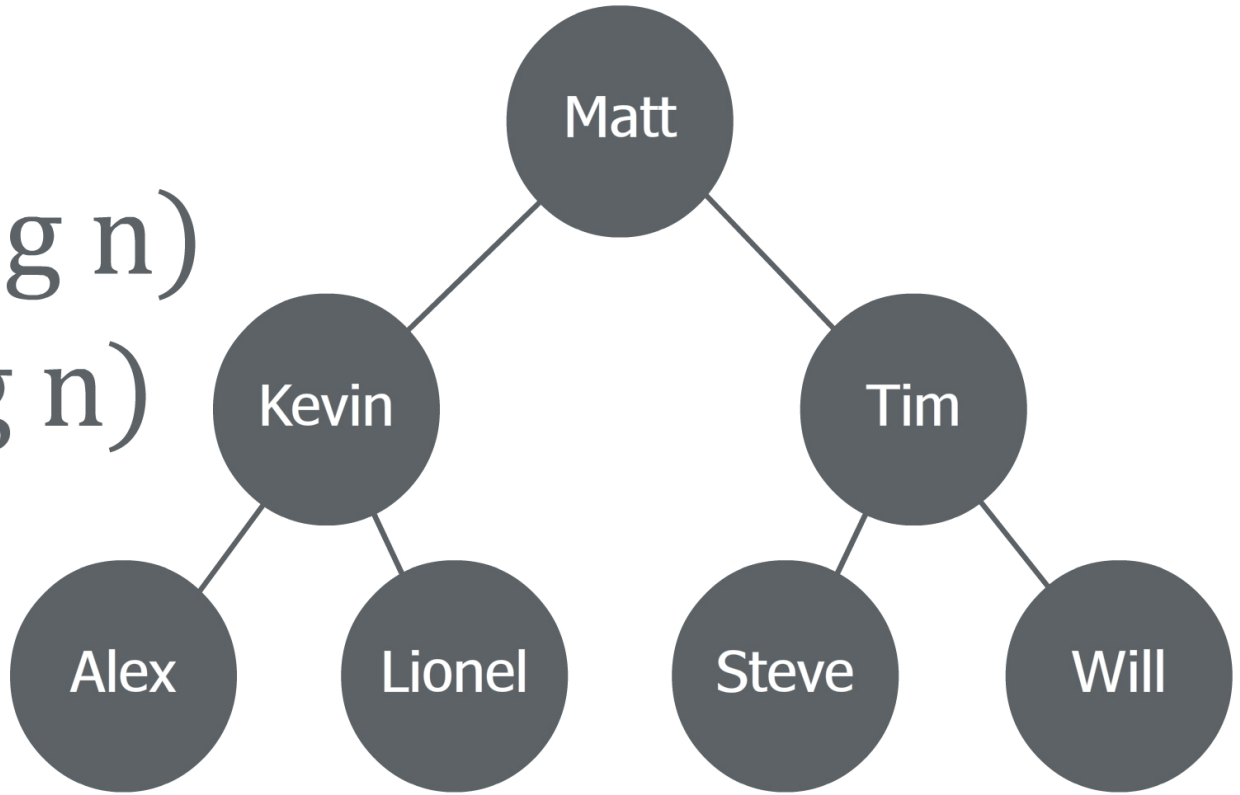
$$2^h = n$$

$$h = \log n$$

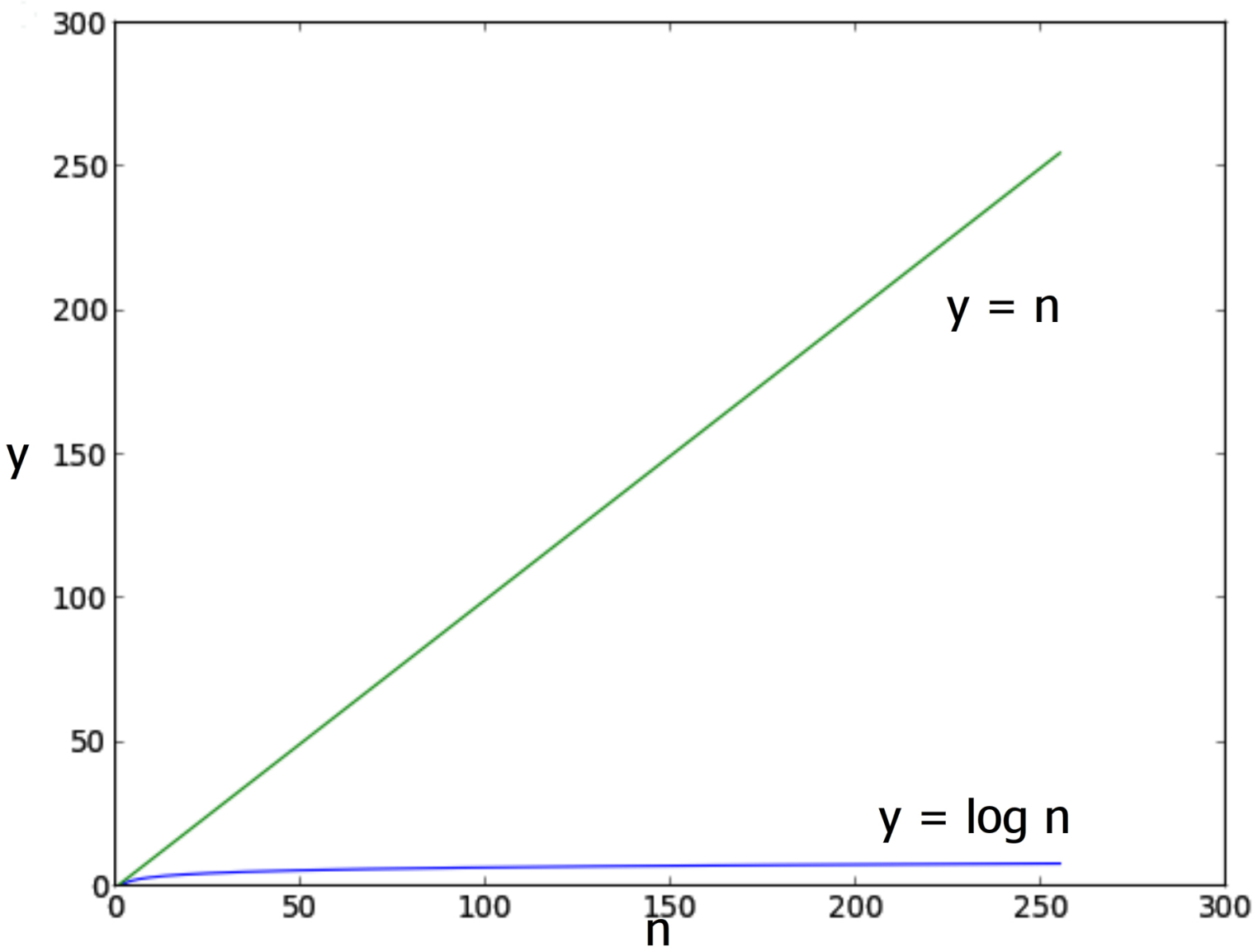
search: $O(\log n)$

insert: $O(\log n)$

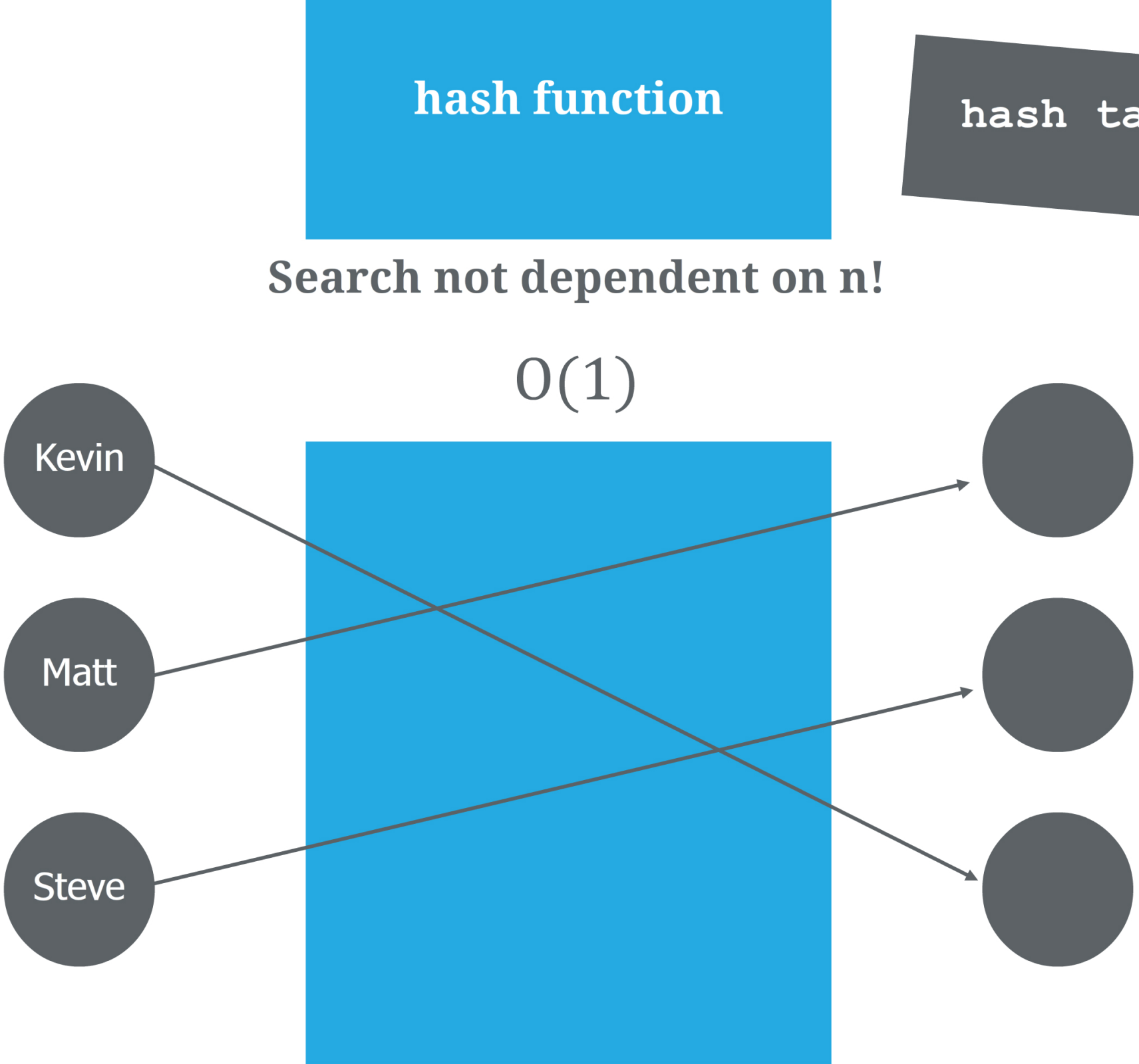
binary search tree



HUGE difference in speed!



$$\log(33554432) = 25$$


hash function
hash table
Search not dependent on n!
O(1)
Kevin
Matt
Steve

sets

Like a hash table table only **just keys** instead of **key, value** pairs

Still $0(1)$ for access, delete

Quiz: traverse a list and remove duplicates...

```
def remove_dupes(a_list):
    cleaned = []
    seen = set ()
    for item in a_list:
        if not item in seen:
            seen.add (item)
            cleaned.append (item)
    cleaned
```

if n = 1e9, CPU can do 1e9 ops/sec ==> **~1 sec. vs ~ 31 years!***

hash tables for everything?

only if you want FAST access to your data:

* caches,
* associative arrays,
* implementing objects (python, ruby),
* implementing sets

Hash tables are so great, I want to make them available as a network service!

April 2009: redis (www.redis.io)

* written by Salvatore Sanfilippo
* a key-value store
* simplest working solution for a networked hash table
* fast. 150k GET/SET per second on i7 laptop
* March 2010: VMware sponsored

redis is simple

Command line interface

```
shell> redis-cli
redis 127.0.0.1:6379> SET lunch good
"OK"
redis 127.0.0.1:6379> GET lunch
"good"
```

Python - all commands are methods

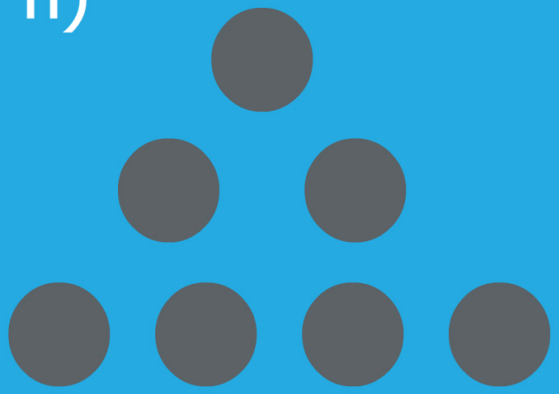
```
import redis
r = redis.StrictRedis (host='localhost')
r.set ('lunch', 'good')
print r.get ('lunch')  # prints "good"
```

where does it fit?

Database cache

without an index your database table behaves like a list (more or less)

with an index your database table responds like a binary search tree (log n)



redis:

* faster and more consistent than an index
* can cache queries that it doesn't make sense to index
* more flexible

Your Stuff is safe

* can log every operation to append-only file - fast

  * fsync at every second (default) or every operation

* trivially simple master - slave replication

Its a **data structure** server!

sets:

```
SADD, SREM, SISMEMBER
```

lists:

```
LPOP, LPUSH, RPOP, RPUSH,
LLEN,

BLPOP, BRPOP
```

Shut up and take my money!

* sorted sets!
* publish - subscribe!
* set expiration for keys!
* increment / decrement!
* hashes (one key, many values)!

now where does it fit ?

when you need FAST, distributed access to data:

* web session management (hash: one key: many values)
* application cache
* as an internal service of transient data (eg. rate limiter)

when you want to leverage its simple treatment of useful design patterns

* key - value store
* publish subscribe
* message broker

Worked Example: A Message Queue

```
def consumer ():
    r = redis.StrictRedis ()

    while True:
        qname, msg = r.blpop ('hello_queue')
        print 'received', msg
```

```
def producer (name):
    r = redis.StrictRedis (),
    i = 0

    while True:
        m = 'hi from {0}! msg #{1}'.format (name, i)
        i += 1

        r.rpush ('hello queue', m)
        time.sleep (5)
```

try more than one producer and consumer!

Worked Example 2 : Pub - Sub

```
def subscriber (channel):
    pubsub = redis.StrictRedis ().pubsub ()
    pubsub.subscribe (channel)

    for msg in pubsub.listen ():
        if msg['type'] == 'message':
            print 'received', msg['data'], 'on', msg['channel']
```

```
def publisher (name, channel):
    r = redis.StrictRedis (),
    i = 0

    while True:
        m = 'hi from {0}! msg #{1}'.format (name, i)
        i += 1

        r.publish (channel, m)
        time.sleep (5)
```

FINDAWAY WORLD