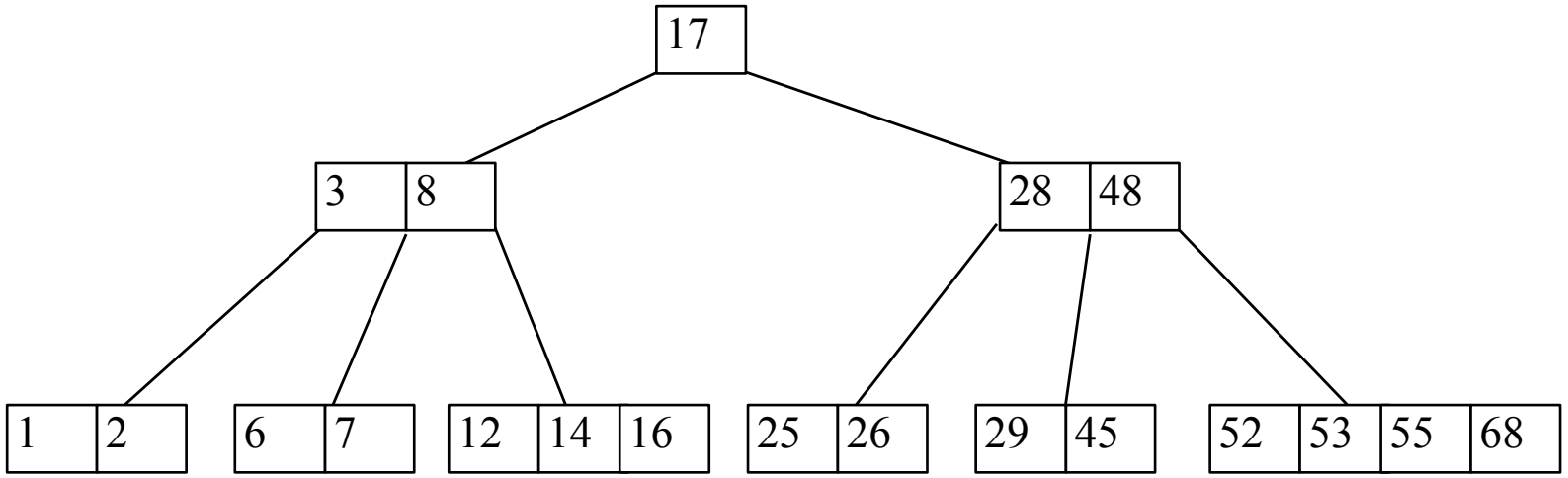
Announcements

MP5 av, due 11/1, 11:59p.

B-tree of order *m* is an *m*-way tree

- For an internal node, # keys = #children -1
- All leaves are on the same level
- All leaves hold no more than *m*-1 keys
- All non-root internal nodes have between $\lceil m/2 \rceil$ and m children
- Root can be a leaf or have between 2 and *m* children.
- Keys in a node are ordered.



Analysis of B-Trees (order *m*)

The height of the B-tree determines the number of disk seeks possible in a search for data.

We want to be able to say that the height of the structure and thus the number of disk seeks is no more than _____.

As we saw in the case of AVL trees, finding an upper bound on the height (given n) is the same as finding a lower bound on the number of keys (given h).

We seek a relationship between the height of the structure (h) and the amount of data it contains (n).

Analysis of B-Trees (order *m*)

We seek a relationship between the height of the structure (h) and the amount of data it contains (n).

- The minimum number of *nodes* in each level of a B-tree of order *m*:
  (For your convenience, let t = _____.)

  root

  level 1

  level 2

  . . .

  level h

- The total number of nodes is the sum of these:

- So, the least **total** number of *keys* is:

Analysis of B-Trees (order *m*)

We seek a relationship between the height of the structure (h) and the amount of data it contains (n). (continued…)

- So, the least **total** number of *keys* is:

- rewrite as an inequality about n, the total number of keys:

- rewrite **that** as an inequality about h, the height of the tree (note that this bounds the number of disk seeks):

BTree Summary

- Goal: Minimize the number of reads from disk
- Build a tree that uses 1 disk block per node
  - Disk block is the fundamental unit of transfer
- Nodes will have more than 1 key
- Tree should be balanced and shallow
  - In practice branching factors over 1000 often used

B-Tree search:

O(m) time per node

O($\log_m$ n) height implies O(m $\log_m$ n) total time

BUT:

(Insert and Delete have similar stories.)

What you should know:

Motivation, Definition, Search algorithm and analysis

Hashing - using “hash tables” to implement _____

Suppose we have the following info…

Locker Number	Name
103	Jay Hathaway
92	Linda Stencel
330	Bonnie Cook
46	Rick Brown
124	Kim Petersen
…	…

…and we want to be able to retrieve a name, given a locker number.

Now suppose our keys are not so nicely described…

Course Number -> Schedule info

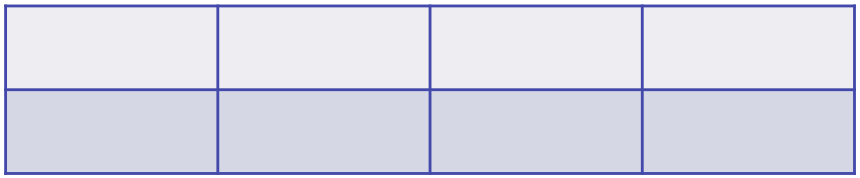
Color -> BMP

Vertex -> Set of incident edges

Flight number -> arrival information

URL -> html page

dice roll -> payoff amt



Some general vocabulary

A *dictionary* is a structure supporting the following:

```
void insert(kType & k, dType & d)
void remove(kType & k)
dType find(kType & k)
```

An *associative array* is a dictionary w a particular interface–

Overloads the [ ] operator for insert and find:

```
myDictionary[“Miguel”]  = 22;
dType d = myDictionary[“Miguel”];
```


Hashing:

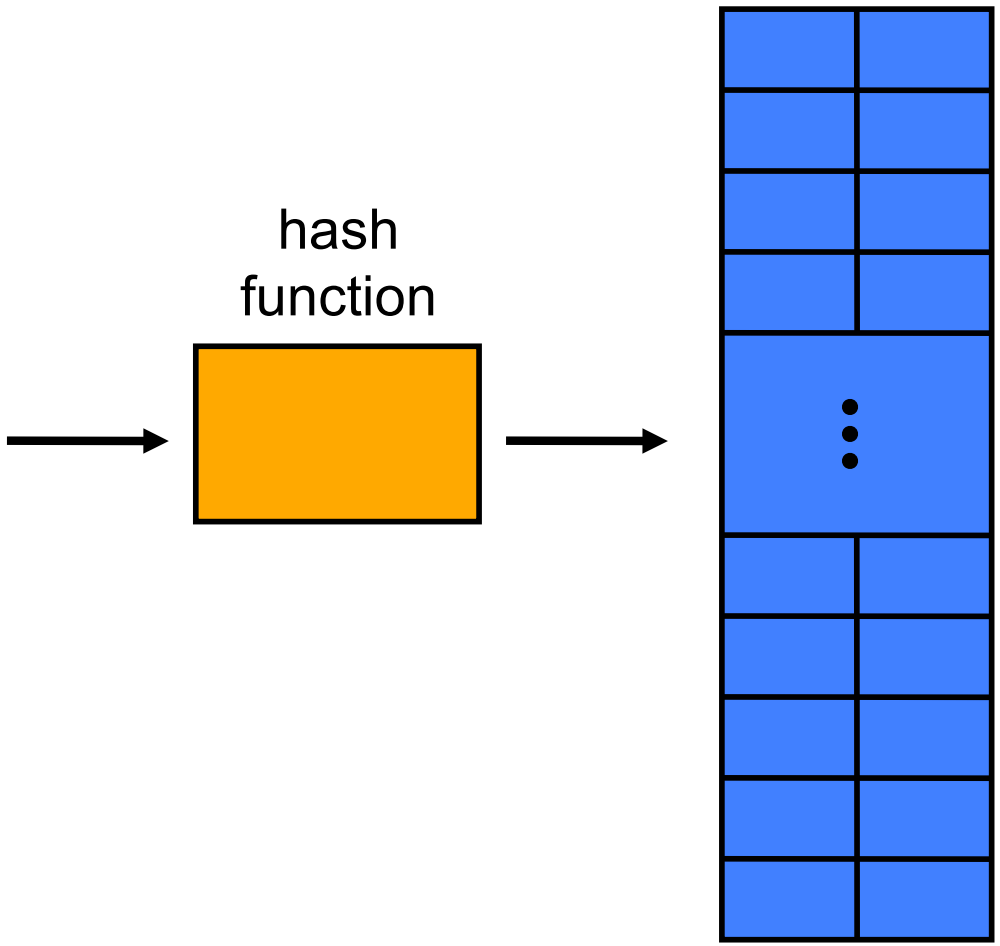
(defn) Keyspace – a (mathematical) description of the keys for a set of data.

Goal: use a function to map the keyspace into a small set of integers.

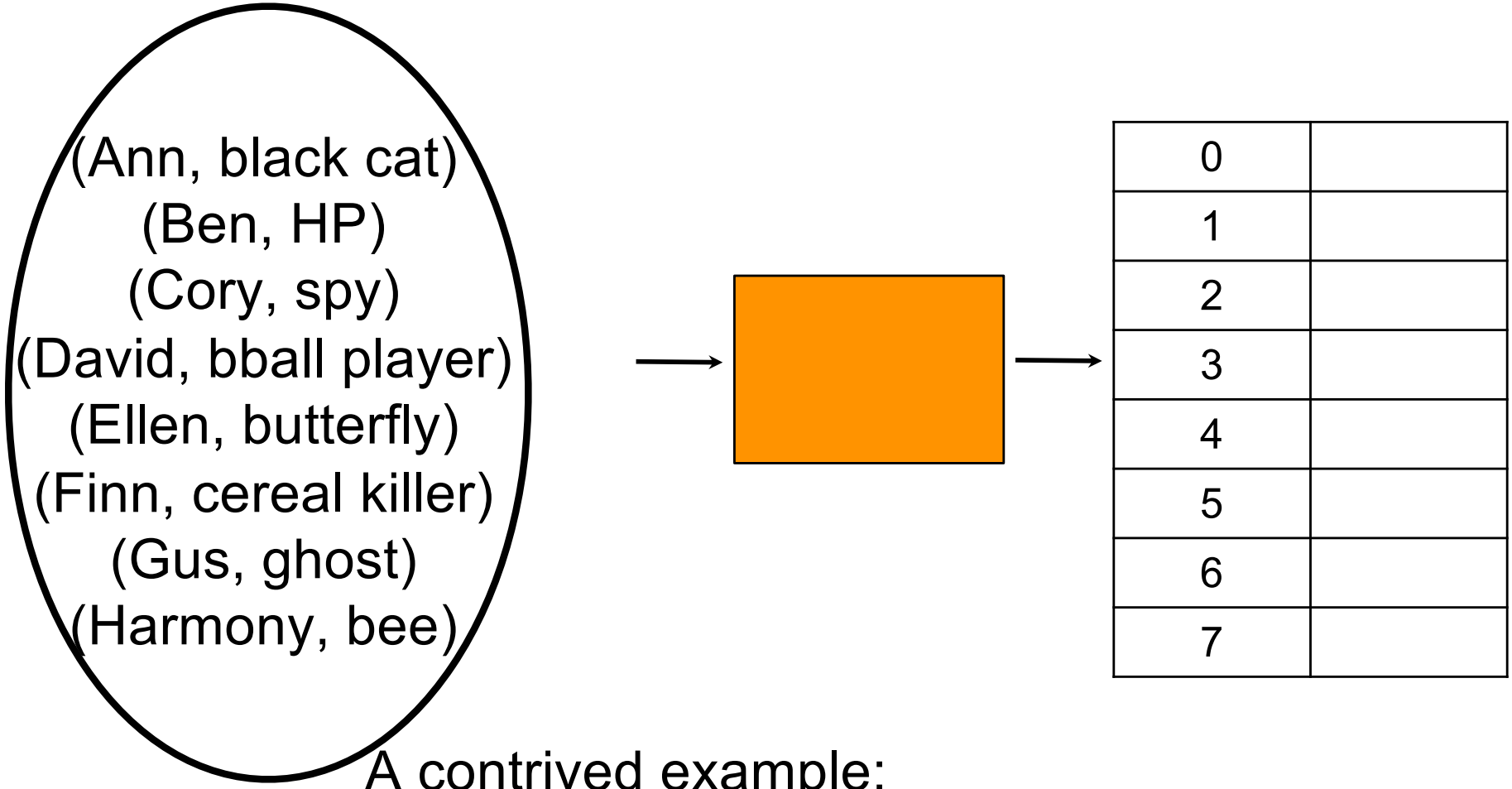
What's fuzzy about this goal?

Problem: Keyspaces are often large…

Basic Idea: we seek a mapping, `h(k)`



A perfect hash function:



A contrived example:

these keys have a fabulous hash fn.

a. each key hashes to a different int

b. collection of keys hash to a seq of ints