Searching

Needles and Haystacks

Problem

You have a long list of things. You occasionally need to retrieve some element of that list, and want to do so quickly and efficiently.

You need to consider…

- How long it takes to add something to your list
- How long it takes to retrieve something from your list
- How long it takes to delete something from your list

Needles: Terminology

- Searching in a list of similar objects
- Each object is composed of a “key” and a “value”
- The “key” is the part of the object that we are searching for
    - Name in an account object
    - time/date in a purchase record
    - etc.
- The “value” is the rest of the information in that object

Solution 1 : Put items in an Array

- If I keep track of highest index so far, and I have room in my array, adding a new element takes about 3 instructions, no matter how big the array is
- If I am searching by array index, then I can find an item with a single instruction
- If I delete an item, I need to move all items below it, so if I have n elements in my array, takes on average $n/2$ moves

Method	Insertion	Search	Deletion
Array key=index	O(1)	O(1)	O(n)

What if index is not the key?

- Suppose I have an array of accounts, and I want to find all accounts owned by a specific named owner
- Inserting a new account takes about 3 instructions
- If there are *n* accounts, takes *n* comparisons to find all accounts for a specific owner
  - Number of instructions per compare varies depending on name length and comparison technique
- Deleting an account takes about *n*/2 moves

Method	Insertion	Search	Deletion
Array key=index	O(1)	O(1)	O(n)
Array key!=index	O(1)	O(n)	O(n)

What if the list is sorted by name?

- Can no longer insert at the end… insertion gets much more expensive. First, you have to find out where to insert; then you have to move everything below that point down one: O(n)
- A brute force search (top to bottom) now takes on average $n/2$ compares instead of *n:* O(n)
- Deletion is unchanged: O(n)

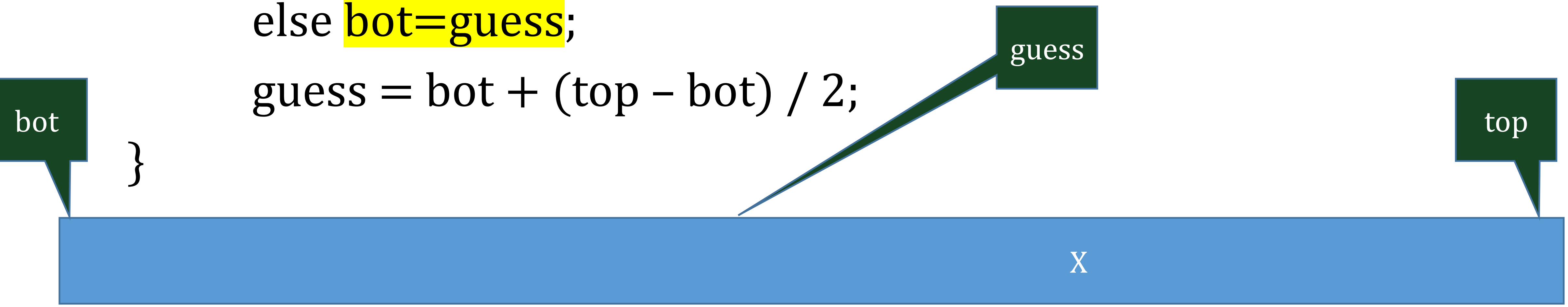
Method	Insertion	Search	Deletion
Array key=index	O(1)	O(1)	O(n)
Array key!=index	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(n) (brute force)	O(n)

Binary Search of Sorted Items

Chapter 14, Section 6.2

- To find x in an array of n sorted items…

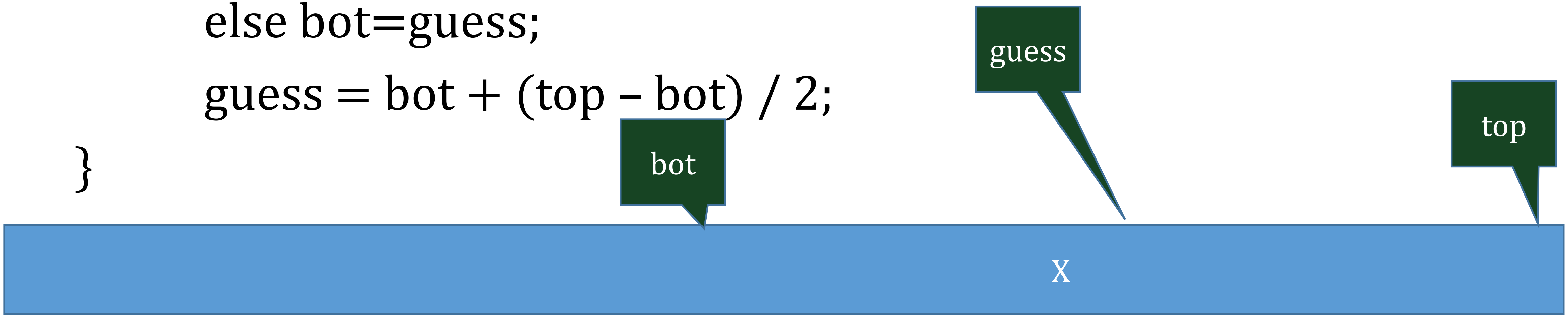
```
int bot=0; int top=n; int guess=n/2;
while(array[guess] != x) {
        if (x < array[guess]) top = guess;
        else bot=guess;
        guess = bot + (top – bot) / 2;
}
```



Binary Search of Sorted Items

- To find x in an array of n sorted items…

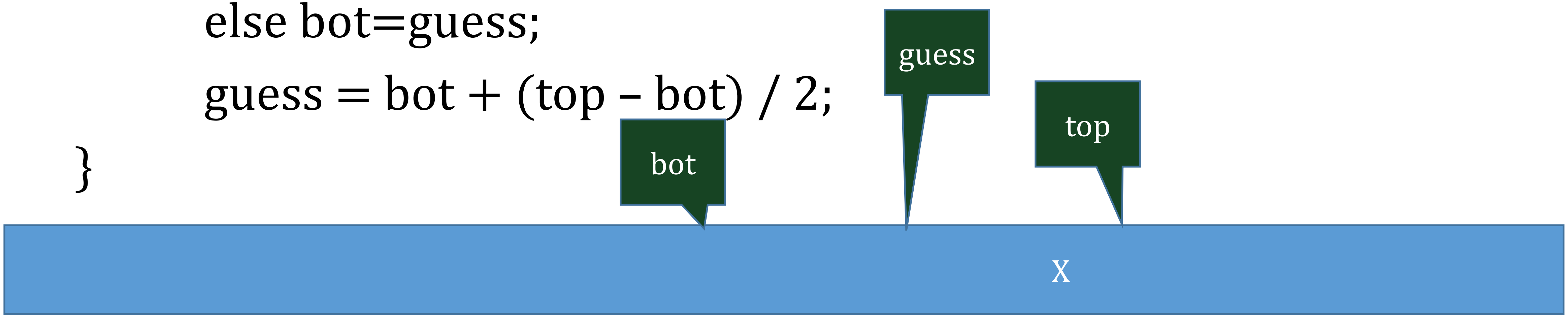
```
int bot=0; int top=n; int guess=n/2;
while(array[guess] != x) {
        if (x < array[guess]) top = guess;
        else bot=guess;
        guess = bot + (top – bot) / 2;
}
```



Binary Search of Sorted Items

- To find x in an array of n sorted items…

```
int bot=0; int top=n; int guess=n/2;
while(array[guess] != x) {
        if (x < array[guess]) top = guess;
        else bot=guess;
        guess = bot + (top – bot) / 2;
}
```



Binary Search performance

- Each iteration divides the size of the list by 2
  - First iteration works on n items, second iteration works on $n/2$ items, Second iteration works on $n/4$ items, …
  - m^{th} iteration works on $n/2^m$ items
- If $n<2^m$ then we must have found x ($n/2^m = 1$)
- Or, $m <= \log_2(n)$

Method	Insertion	Search	Deletion
Array key=index	O(1)	O(1)	O(n)
Array key!=index	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(n) (brute force) O(log n) (bsearch)	O(n)

Binning for Unsorted Items

- Keep two or more bins… lists of objects… bins are a list of lists
- Quick function to determine what bin an element belongs in
- Trick is to equalize binsize… so for m bins, binsize ~= n/m
- Time to insert : find bin, add to bin - O(1)
- Time to search : find bin, search in bin – O(n/m)
- Time to delete : find bin, find in bin, delete – O(n/m)

- More bins mean faster access, but more overhead

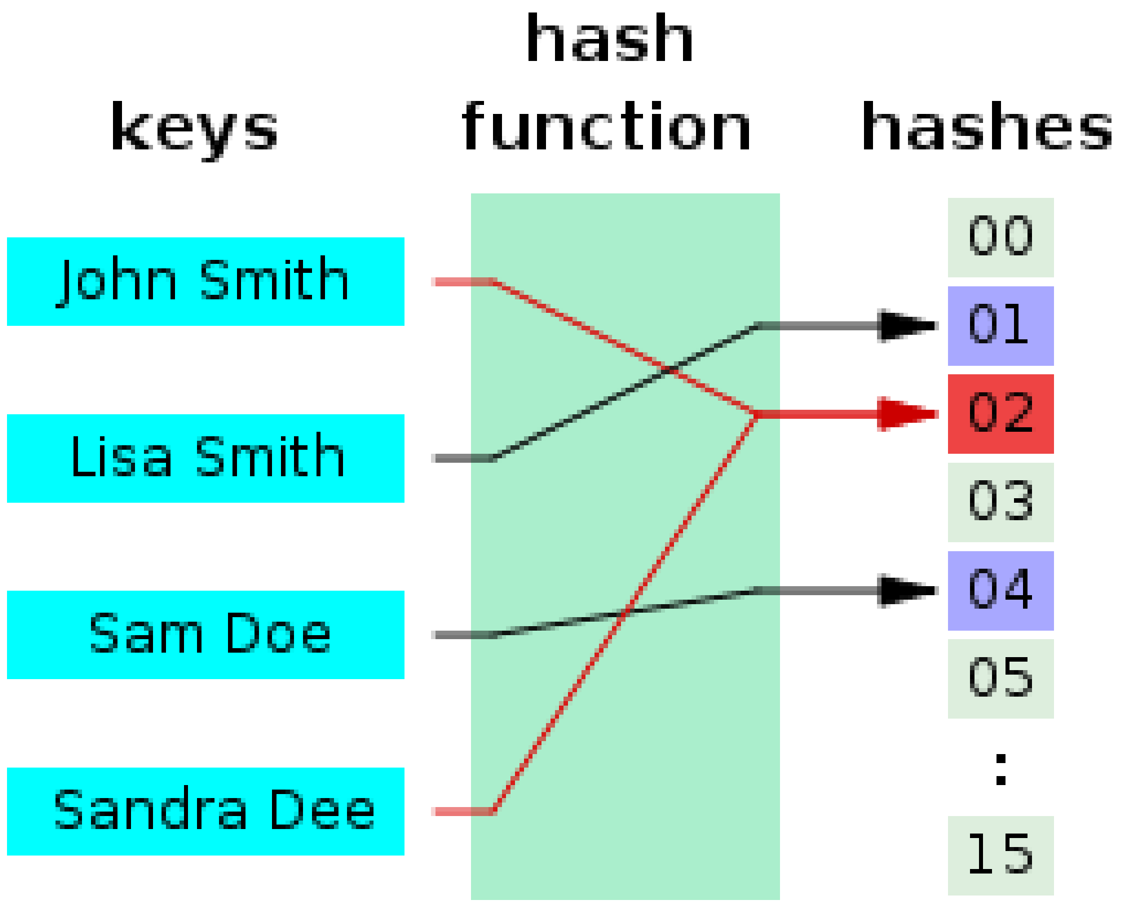
Hashing

- Pick a fixed bin size: "c"
- Choose number of bins, based on the size of the list
  - $m = n/c$
  - If there are approximately equal number of items in each bin, binsize= $\pm n/m = \pm\, n/(n/c) = \pm c$
- Find a hash function: hash(key)=bin_index
  - Guarantee, if (key_1==key_2), then hash(key_1)==hash(key_2) i.e. the same key always goes to the same bin
- Hash collision allowed, but rare (only c times per bin):

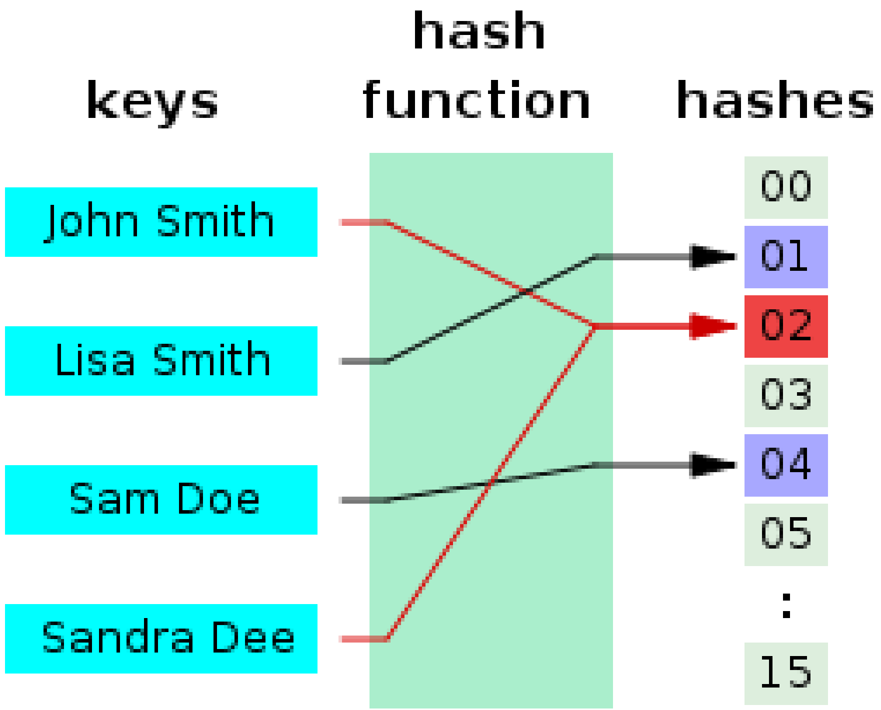
  key_1 != key_2 but hash(key_1)==hash(key_2)

Example Hash

- Translate keys to index 0-15
- Each key hashes to the same index every time
- Multiple keys may map to a single index



Example Hash Table



	Name	Town	ID
0			
1	Lisa Smith	Vestal	6894
2	John Smith	Endicott	1548
	Sandra Dee	Binghamton	6442
3			
4	Sam Doe	Johnson City	2954
5			
…			
15			

Hash Performance

- Insertion: hash function runs quickly, but once we find a bin, we need to insert in that bin. Since binsize=±c, insertion O(c)
- Search: hash function runs quickly, but once we find a bin, we need to search for the key in that bin. Since binsize=±c, search O(c)
- Delete: hash function runs quickly, but once we find a bin, we need to search for the key in that bin. Since binsize=±c, delete O(c)

Method	Insertion	Search	Deletion
Array key=index	O(1)	O(1)	O(n)
Array key!=index	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(n) (brute force) O(log n) (bsearch)	O(n)
Hash Map	O(1)	O(1)	O(1)

Using Hash in Java

- All java objects have a hashCode method: Object → int
- HashMap (concrete Collections "Map" implementation)
  - Guesses at n, chooses m so that binsize is constant and low, c
  - Allocates n/c=m bins
  - Gets bin index by key.hashCode()%m
  - Manages hash collisions for us automatically
- HashMap depends on valid hashCode
  - Spreads objects over integers randomly
  - Equal objects have the same hashcode

Hash problem

- Integer class hashCode method: return value*100;

- HashMap has m=100 (100 bins)

- binIndex = hashCode()%100 = (value*100)%100 = 0
  - All values map to the same bin!!!!

- Solution: hashCode method: return (value*prime)%(max_int)
  - No matter what m is, (value*prime)/m will distribute evenly