Exercises: Special Case Search Structures

Design super-fast search structures for the following special cases:

a) Data consists of *ints* in the range 0 … 100

b) Data consists of *chars* in the range A-Z, a-z, 0-9

c) Data consists of a few dozen Strings, each of which has a unique last character

d) Data consists of a few dozen *ints*, most of which have a unique last two digits

e) Data consists of a few hundred Points, such that if you add the two coordinates modulo 1000 you get a mostly unique number in the range 0 ... 999

Hash Tables

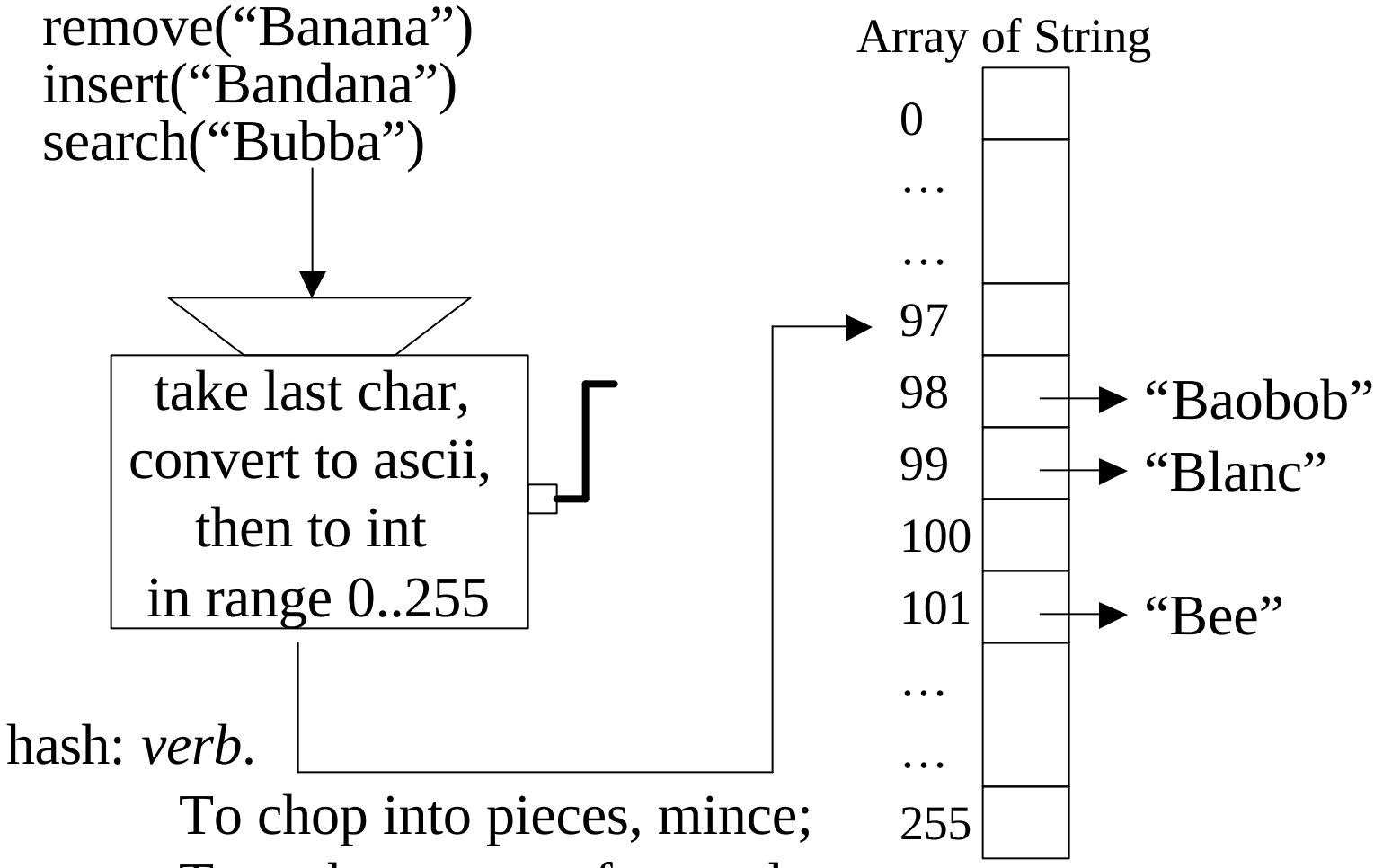
& Dictionaries

Goal

- Arrays are too static
  - Don’t grow dynamically (but can use doubling trick)
  - Hard to insert/remove in the middle
  - Index has to be a natural number
- Recursive structures are too dynamic
  - Hard to enforce nice structure (e.g., balanced trees)
  - Complicated code
- Compromise:
  - Use an array most of the time
  - Allow arbitrary key for indexing
  - Make special cases for inserting/removing, if needed

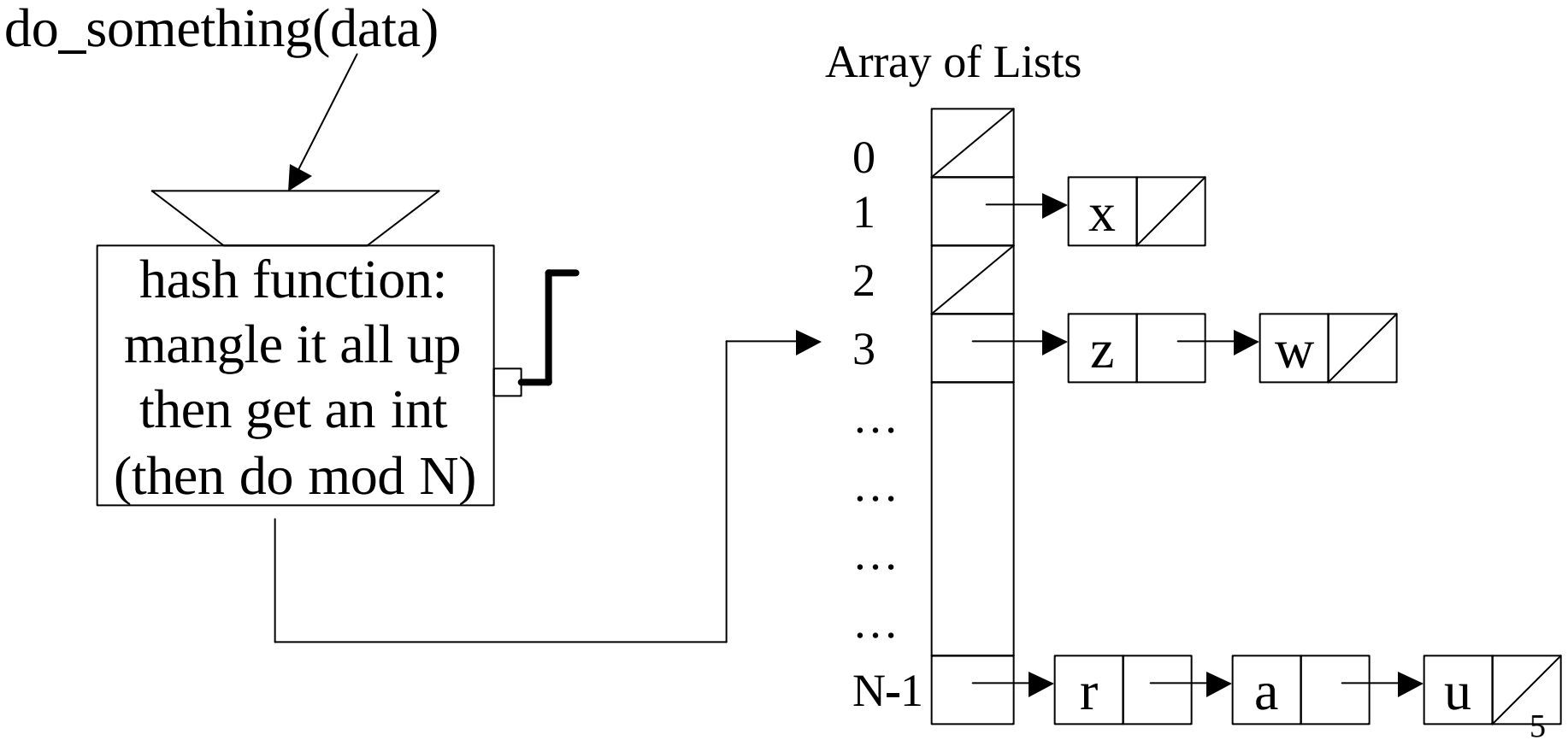
Arrays using arbitrary keys

c) Data consists of a few dozen Strings, each of which has a unique last character



Hash Tables

- Generalized:
  - need “hash function” for data objects
  - need linked lists (or something) for dealing with collisions



Hash Table Algorithms

- Need a *hash function* that can convert from data objects (the “key”) to int (the “bucket number”)
  - Call it the *hash code* for the object
- Insert(obj):
  - compute hash code for obj
  - Append obj to list at that bucket
- Search(obj):
  - compute hash code for obj
  - Look for obj in list at that bucket
- Remove(obj):
  - compute hash code for obj
  - Remove obj from list at that bucket

Performance

- Wildly optimistic assumption:
  - Assume all hash codes turn out to be unique and in the range 0 … N-1
- Insert: O(1)
- Remove: O(1)
- Delete: O(1)

Neat trick, huh?

Key Issues

- What should the hash function be?
- How big should the array be?
- How should we organize the linked lists?

Hash Functions

- Goal:
  - Every (different) object should get a different hash code
- Realistic Goal:
  - Different objects should be likely to get different hash code
  - *Collisions* should be more or less random
  - Use the whole range 0 … N-1, evenly

Hash Functions: 1st attempts

- Data is student id numbers:
  - hash(ID) = 55 (for any ID)
    - bad → hash table reverts to linked list behavior
  - hash(ID) = two most significant digits
    - bad → all data ends up in just a few buckets
  - hash(ID) = two least significant digits
    - pretty good → collisions happen sort of randomly (373333 and 375933 collide)
  - hash(ID) = add pairs of digits, mod 100
    - e.g., 375933 → (37 + 59 + 33) mod 100
    - better → even less of a pattern to the collisions
  - hash(ID) = square the number, then take middle digits
    - popular in practice … what is the pattern of collisions?
- Of course, we want it to scale to arbitrary N, though!

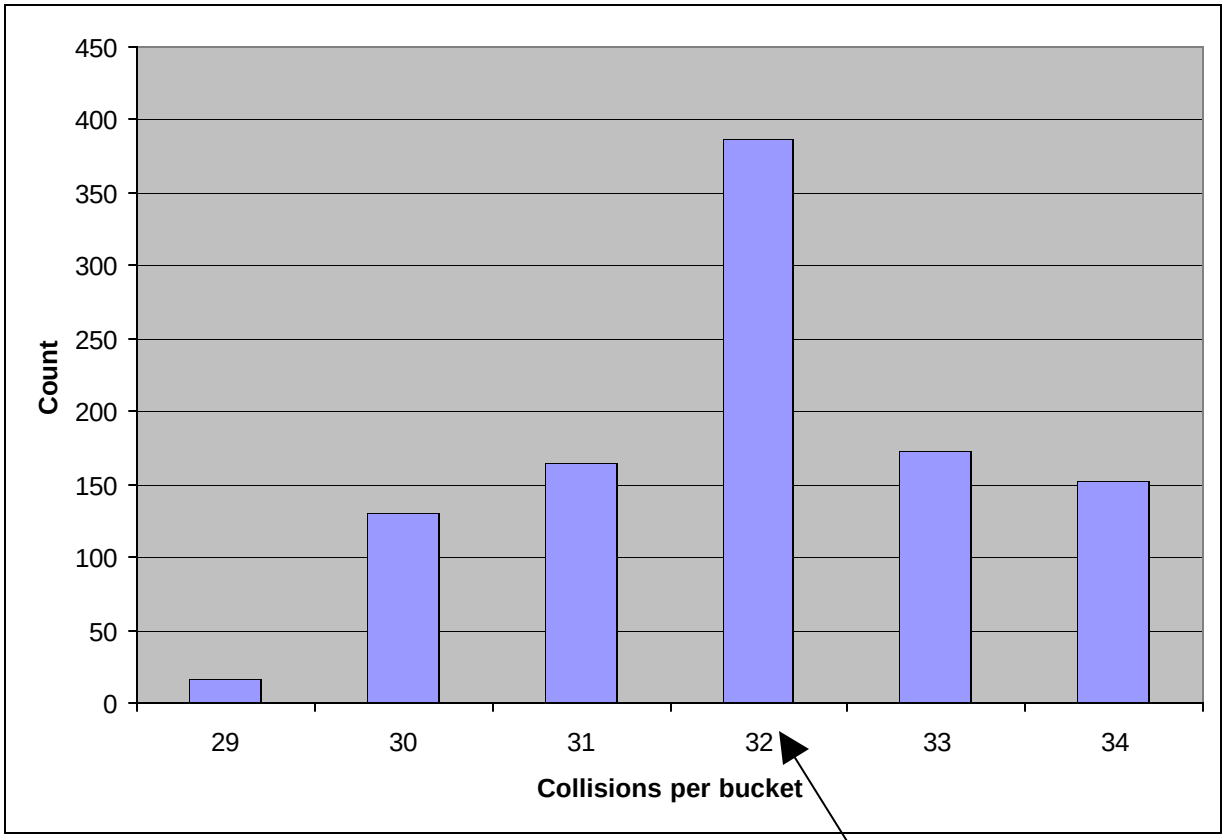
Example: Multiplicative Hash Function

- Suppose N is a power of 2 (say N = 1024)
- Suppose keys are ints in range 0 … 32767
- Use H(k) = (((32768*0.6125423371*k)%32768)%1024)
  - Where do these constants come from?
- Try it out:

```
int[] histogram = new int[1024];
for (int k = 0; k < 32768; k++) {
        int bucket = H(k);
        histogram[bucket]++;
}
for (int i = 0; i < 1024; i++)
        System.out.println(i + " " + histogram[i]);
```

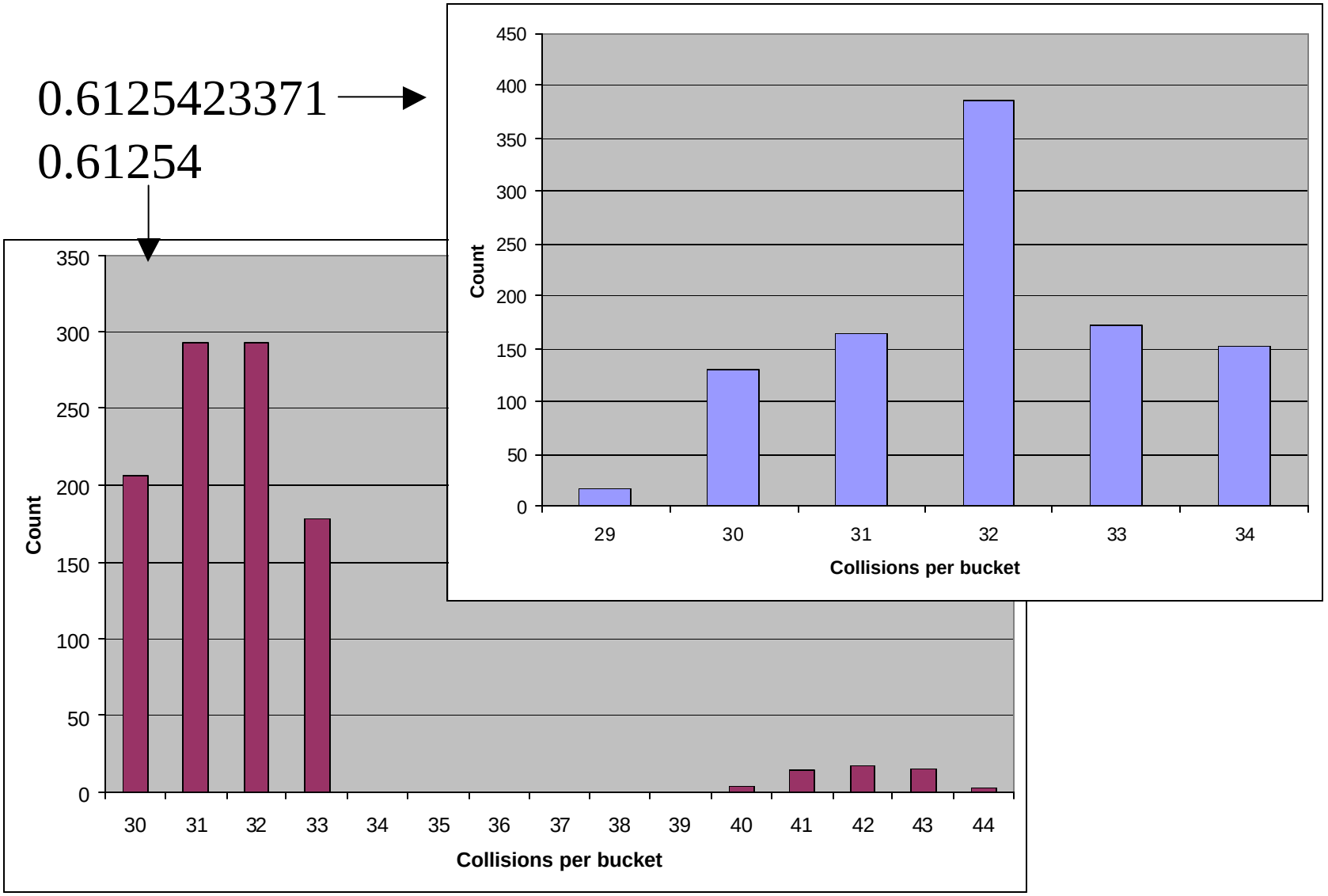
0	34
1	33
2	32
3	32
4	32
5	32
6	31
7	32
8	32
9	34
10	32
11	31
12	30
13	33
14	33
15	32
16	32
17	33
18	32
19	30
20	31
21	33
22	34
23	32
24	32
25	30
26	32
27	32
...	...

bucket 25
had 30 collisions



most buckets
had 32 collisions

What if we use 0.61254 instead?



Hash Functions in Practice

- Need to handle arbitrary object types
  - Strings, Integers, doubles, ChessGames, etc.
  - Each type has a different way of getting an integer
- Need to convert from int to bucket #
- Two step process: for arbitrary object *x*
  - (1) Call *x*.hashCode() to get an integer *k*
  - (2) Use *k % N* as the bucket number

public int hashCode()

- Instance method, defined in java.lang.Object
- Computes the *int* hash code of *this* object
- Specification:
  - Must return same value repeatedly for same object
  - Two objects that *equals* must have the same hash code
  - Otherwise, they should have different hash codes
- Easy examples:
  - Integer, Short, Byte → just returns value as an *int*
  - Float, Double → return bit pattern of value as an *int*

hashCode() for non-primitives

- Strings
  - Compute based on each character (arithmetic) as:
    $s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + ... + s[n-1]$
- Other user-defined types:
  - Usually based on String.hashCode(), Integer.hashCode(), etc.
  - You have to write your own hashCode for your own classes! Obey the contract…
- Default:
  - Dynamic binding resolves to Object.hashCode()
  - Just uses address of object in memory

Perfect Hashing

- If we knew what the keys were ahead of time, we could design a perfect hash function
  - Each key gets a different hash code
  - Hash codes start at 0, count upwards with no gaps
- When is this used?
  - Compiler: keywords are known ahead of time:
    - public = 0, private = 1, protected = 2, void = 3, int = 4, …
    - Hash table lookup is much faster than String.equals
    - Compiler never does String.equals on keywords any more
      - if (keywordTable.search(someword)) { … } else { … }

Hash Table Worst-Case Performance

- If hash codes are terribly chosen:
  - All buckets have 0 objects
  - Except one bucket that has all the objects
  - Revert to linked-list behavior
  - get: O(n)
  - put: O(1)
- If hash codes are perfectly chosen:
  - Each bucket has exactly the same number of objects
    - N buckets, D objects → λ = D/N objects per bucket
  - get: O(λ)
  - put: O(1)

Load Factor: λ

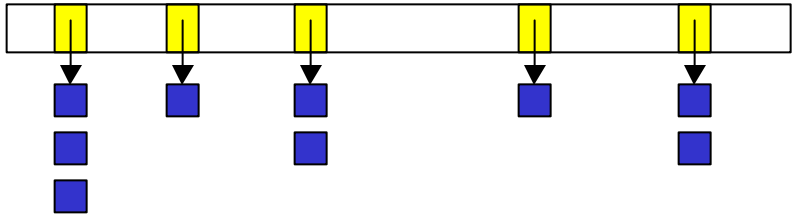
- We usually assume that hash codes are randomly distributed
  - gives close to "perfectly chosen" performance
  - statistics & probability can tell you how many collisions to expect, what the graph would look like, etc.
    - won't get exactly λ = D / N objects per bucket
    - but will get λ ± 33% or so with very high probability
- So performance is:
  - get: O(λ) where λ = D/N = # objects / # buckets
  - put: O(1)
- Want no more than one object per bucket
  - Want λ = 1 so let # buckets = # objects
  - Or better yet, use λ = 0.75
    so let # buckets = #objects * 1.33

Controlling Load Factor

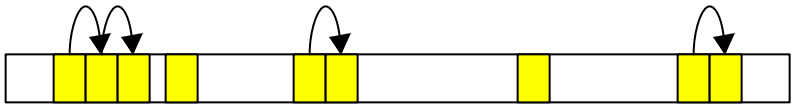
- Want to keep $\lambda = 0.75$, but don't know # objects apriori
- Use array doubling trick again
  - These tricks are used and reused and reused again
- Gives in practice:
  - O(1) performance for all operations

Hash Table Flavors

- So far we have hash tables with separate chaining



- Could do hash table with open-addressing
  - Don't chain, instead try multiple buckets
  - Start with bucket *h*, generate sequence of bucket #'s
- Example: linear probing
  - Start with bucket *h*, then try *h+1*, *h+2*, …



  - Analysis gets tricky, but no extra memory allocation

Hash Table Summary

- Supports SearchStructure interface
  - insert, delete, search all in O(1) “expected” worst case
  - “expected” = if we assume hash codes are random, then with very high probability
- What else can we do with them?

Dictionaries

- Dictionary structure is a set of <*key*, *value*> pairs
- Want fast search for *value* given a *key*
- Easy with existing search structure interface:
  - Define a class Pair to store *key* and *value*
  - Define *equals* that only compares *key* half of two pairs
  - Define *compareTo* that only compares *key* half of pairs
    - Used in the BST search structure
  - Define *hashCode* that only hashes *key* half of a pair
    - Used in the hash table search structure
- Better yet, modify BST and hash table to maintain *key* and *value*
  - Will use inner classes *BST.KeyValuePair* and *HashTable.KeyValuePair*

HashSet and HashMap

- Java provides HashSet as a sequence structure
  - Uses separate chained hashing, as we did above
  - Aims for load factor $\lambda = 0.75$, and doubles size when exceeded, as we did
  - Uses *x.hashCode()* instance method, as we did
- Java provides HashMap as a dictionary structure
  - Similar, but maintains *key* and *value* pairs
  - Slightly different interface:
    - search by key (fast)
    - search by value (slow)
    - enumerate keys
    - enumerate pairs
    - enumerate values
    - etc.