CSE332: Data Abstractions

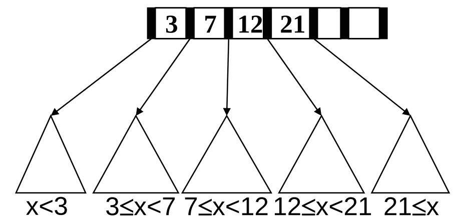
Lecture 10: More B Trees; Hashing

Dan Grossman

Spring 2010

B+ Tree Review

- M-ary tree with room for L data items at each leaf
- Order property:
  Subtree **between** keys *x* and *y* contains only data that is ≥ ***x*** and < ***y*** (notice the ≥)
- Balance property:
  All nodes and leaves at least half full, and all leaves at same height
- **`find`** and **`insert`** efficient
  - **`insert`** uses *splitting* to handle overflow, which may require splitting parent, and so on recursively

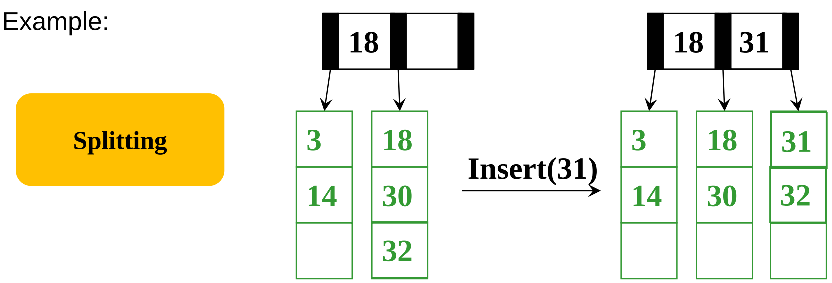


Can do a little better with insert

Eventually have to split up to the root (the tree will fill)

But can sometimes avoid splitting via *adoption*
- Change what leaf is correct by changing parent keys
- This idea "in reverse" is necessary in deletion (next)

Example:

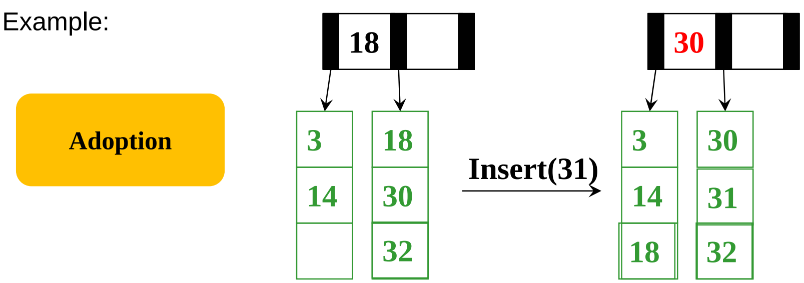


Adoption for insert

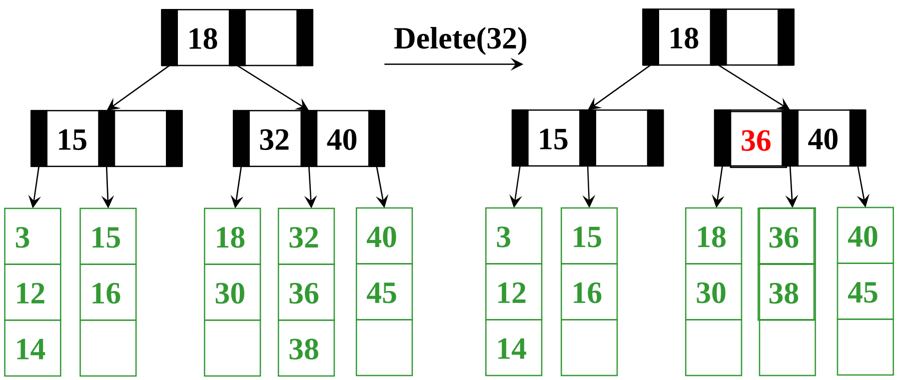
Eventually have to split up to the root (the tree will fill)

But can sometimes avoid splitting via *adoption*
- Change what leaf is correct by changing parent keys
- This idea "in reverse" is necessary in deletion (next)

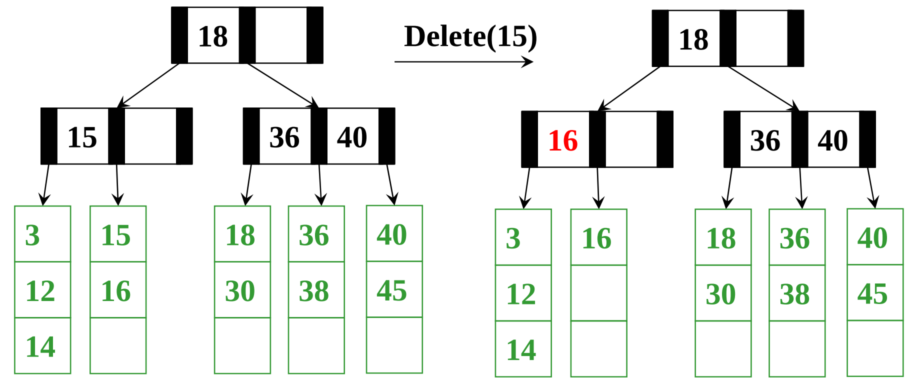
Example:



And Now for Deletion…



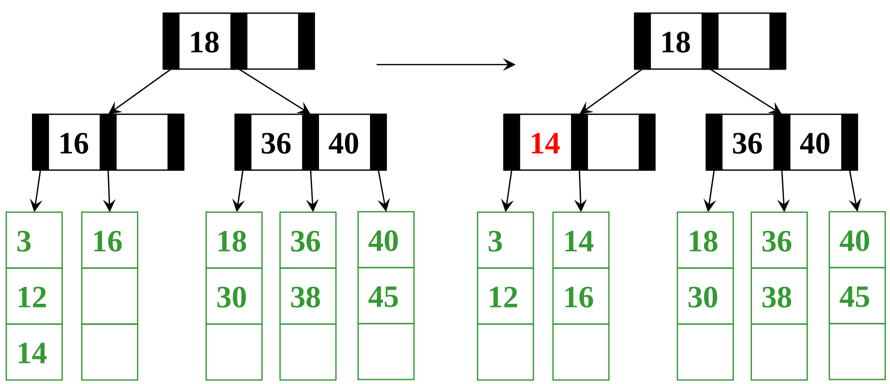
M = 3 *L* = 3



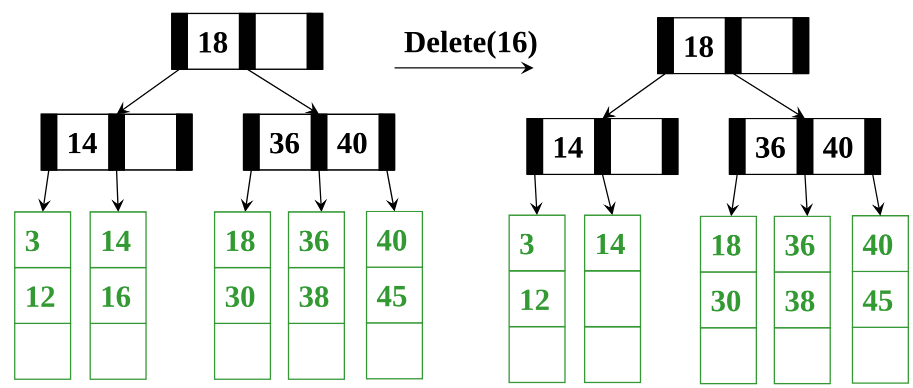
What's wrong?

Adopt from a neighbor!

M = 3 *L* = 3

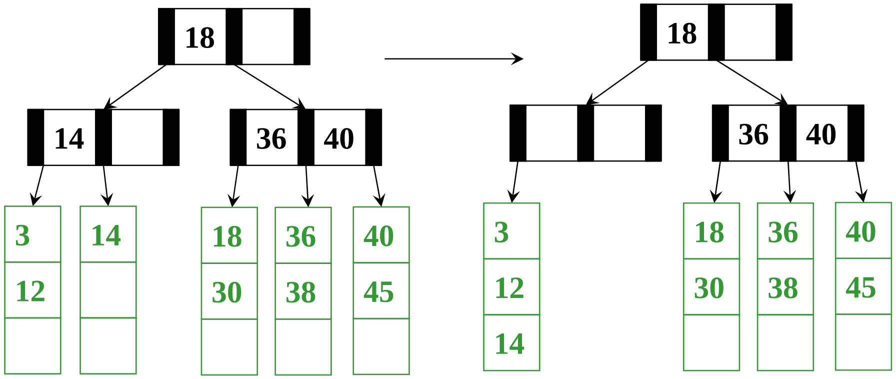


M = 3 *L* = 3



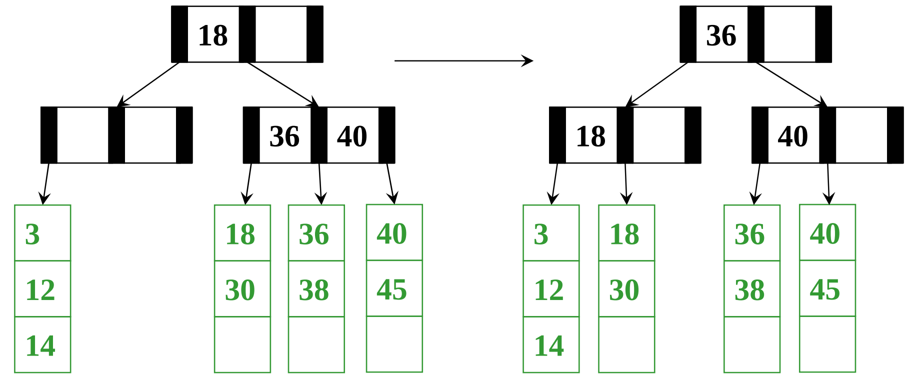
Uh-oh, neighbors at their minimum!

M = 3 *L* = 3

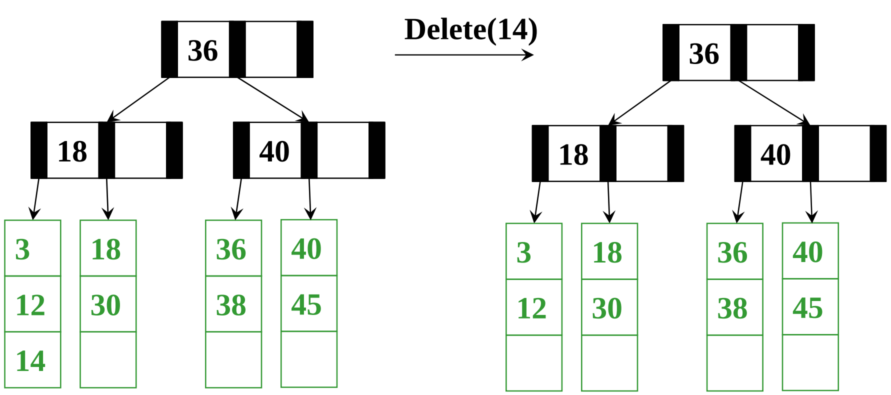


Move in together and remove leaf – now parent might underflow; it has neighbors

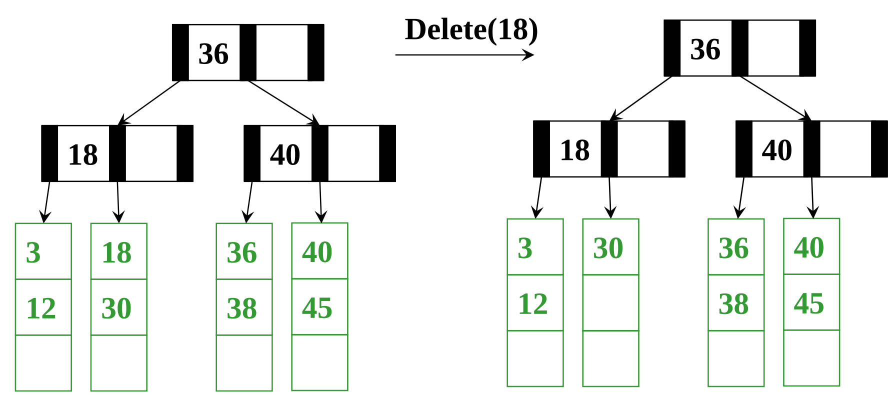
M = 3 *L* = 3



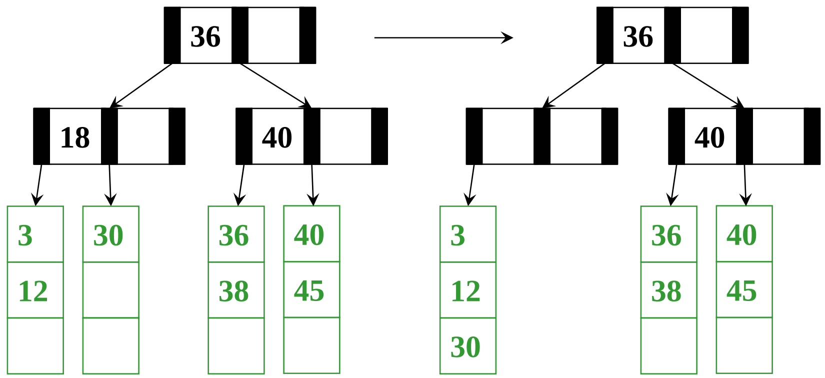
M = 3 *L* = 3



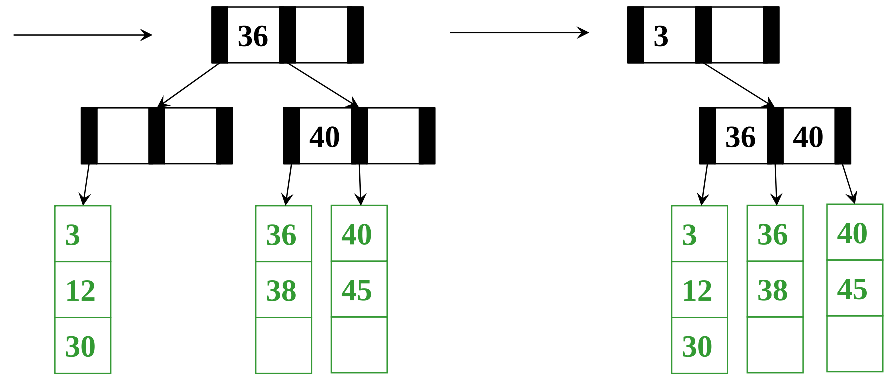
M = 3 *L* = 3



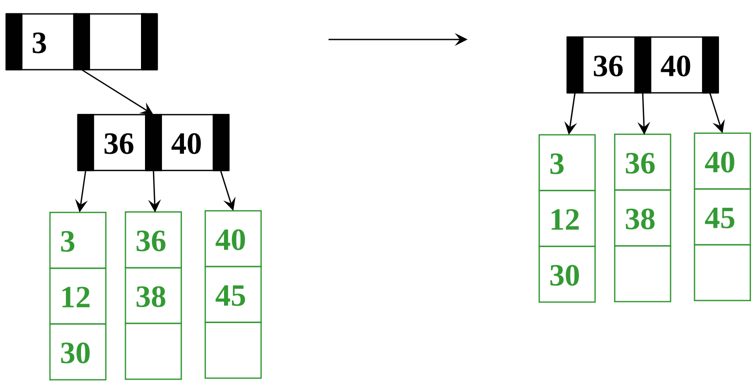
M = 3 *L* = 3



M **= 3** ***L*** **= 3**



M **= 3** ***L*** **= 3**



M **= 3** ***L*** **= 3**

Deletion Algorithm

1. Remove the data from its leaf

2. If the leaf now has $\lceil L/2 \rceil - 1$, *underflow!*
   - If a neighbor has > $\lceil L/2 \rceil$ items, *adopt* and update parent
   - Else *merge* node with neighbor
     - Guaranteed to have a legal number of items
     - Parent now has one less node

3. If step (2) caused the parent to have $\lceil M/2 \rceil - 1$ children, *underflow!*
   - …

Deletion algorithm continued

3. If an internal node has $\lceil M/2 \rceil - 1$ children
   - If a neighbor has > $\lceil M/2 \rceil$ items, *adopt* and update parent
   - Else *merge* node with neighbor
     - Guaranteed to have a legal number of items
     - Parent now has one less node, may need to continue up the tree

If we merge all the way up through the root, that's fine unless the root went from 2 children to 1
- In that case, delete the root and make child the root
- This is the only case that decreases tree height

Efficiency of delete

- Find correct leaf: $O(\log_2 M \log_M n)$
- Remove from leaf: $O(L)$
- Adopt from or merge with neighbor: $O(L)$
- Adopt or merge all the way up to root: $O(M \log_M n)$

Total: $O(L + M \log_M n)$

But it's not that bad:
- Merges are not that common
- Remember disk accesses were the name of the game: $O(\log_M n)$

Note: Worth comparing insertion and deletion algorithms

B Trees in Java?

For most of our data structures, we have encouraged writing high-level, reusable code, such as in Java with generics

It is worth knowing enough about "how Java works" to understand why this is probably a bad idea for B trees
- Assuming our goal is efficient number of disk accesses
- Java has many advantages, but it wasn't designed for this
- If you just want a balanced tree with worst-case logarithmic operations, no problem
  - If *M*=3, this is called a 2-3 tree
  - If *M*=4, this is called a 2-3-4 tree

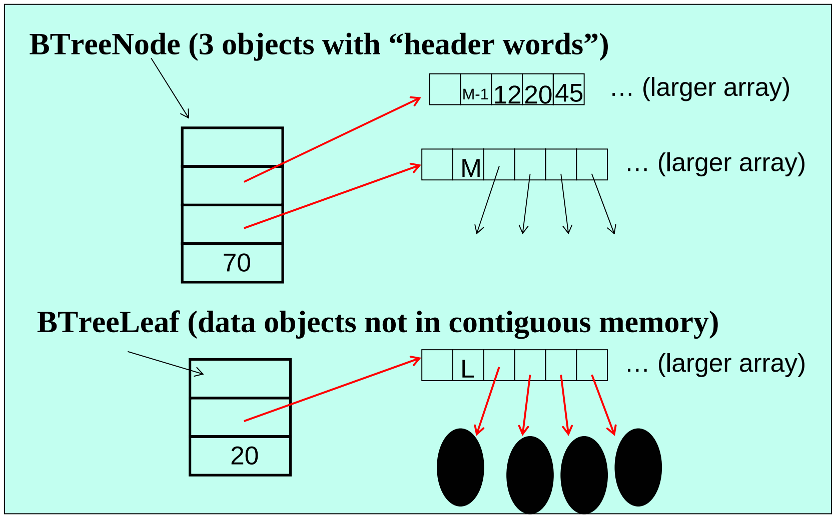
The key issue is extra *levels of indirection*…

Naïve approaches

Even if we assume data items have **`int`** keys, you cannot get the data representation you want for "really big data"

```
interface Keyed<E> {
  int key(E);
}
class BTreeNode<E implements Keyed<E>> {
  static final int M = 128;
  int[] keys = new int[M-1];
  BTreeNode<E>[] children = new BTreeNode[M];
  int numChildren = 0;
  …
}
class BTreeLeaf<E> {
  static final int L = 32;
  E[] data = (E[])new Object[L];
  int numItems = 0;
  …
}
```

What that looks like



The moral

- The whole idea behind B trees was to keep related data in contiguous memory
- All the red references on the previous slide are inappropriate
  - As minor point, beware the extra "header words"
- But that's "the best you can do" in Java
  - Again, the advantage is generic, reusable code
  - But for your performance-critical web-index, not the way to implement your B-Tree for terabytes of data
- C# may have better support for "flattening objects into arrays"
  - C and C++ definitely do
- Levels of indirection matter!

Possible "fixes"

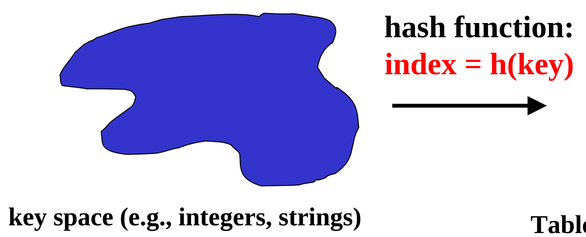
- Don't use generics
  - No help: all non-primitive types are reference types
- For internal nodes, use an array of pairs of keys and references
  - No, that's even worse!
- Instead of an array, have *M* fields (key1, key2, key3, …)
  - Gets the flattening, but now the code for shifting and binary search can't use loops (tons of code for large *M*)
  - Similar issue for leaf nodes

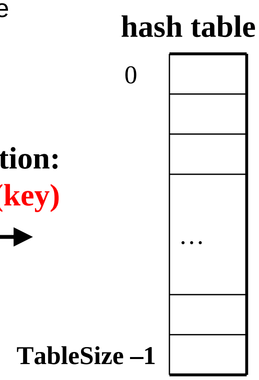
Conclusion: Balanced Trees

- *Balanced* trees make good dictionaries because they guarantee logarithmic-time **`find`**, **`insert`**, and **`delete`**
  - Essential and beautiful computer science
  - But only if you can maintain balance within the time bound
- AVL trees maintain balance by tracking height and allowing all children to differ in height by at most 1
- B trees maintain balance by keeping nodes at least half full and all leaves at same height
- Other great balanced trees (see text; worth knowing they exist)
  - Red-black trees: all leaves have depth within a factor of 2
  - Splay trees: self-adjusting; amortized guarantee; no extra space for height information

Hash Tables

- Aim for constant-time (i.e., *O*(1)) **`find`**, **`insert`**, and **`delete`**
  - "On average" under some reasonable assumptions
- A hash table is an array of some fixed size
- Basic idea:





Hash tables

- There are *m* possible keys (*m* typically large, even infinite) but we expect our table to have only *n* items where *n* is much less than *m* (often written *n* << *m*)

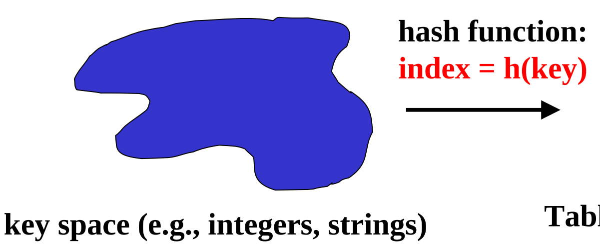
Many dictionaries have this property

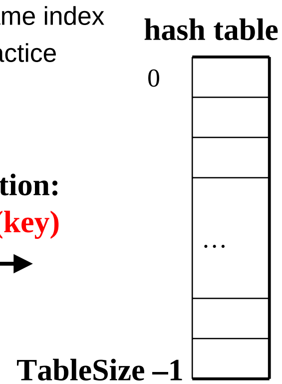
- Compiler: All possible identifiers allowed by the language vs. those used in some file of one program
- Database: All possible student names vs. students enrolled
- AI: All possible chess-board configurations vs. those considered by the current player
- …

Hash functions

An ideal hash function:

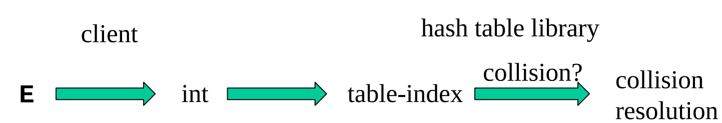
- Is fast to compute
- "Rarely" hashes two "used" keys to the same index
  - Often impossible in theory; easy in practice
  - Will handle *collisions* in next lecture





Who hashes what?

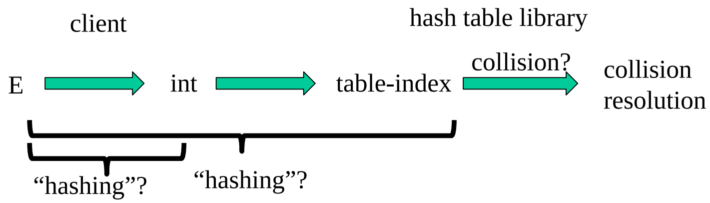
- Hash tables can be generic
  - To store elements of type **`E`**, we just need **`E`** to be:
    1. Comparable: order any two **`E`** (like with all dictionaries)
    2. Hashable: convert any **`E`** to an **`int`**
- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:



- We will learn both roles, but most programmers "in the real world" spend more time as clients while understanding the library

More on roles

Some ambiguity in terminology on which parts are “hashing”



Two roles must both contribute to minimizing collisions (heuristically)

- Client should aim for different ints for expected items
  - Avoid “wasting” any part of **`E`** or the 32 bits of the **`int`**
- Library should aim for putting “similar” **`int`**s in different indices
  - conversion to index is almost always “mod table-size”
  - using prime numbers for table-size is common

What to hash?

In lecture we will consider the two most common things to hash: integers and strings

- If you have objects with several fields, it is usually best to have most of the “identifying fields” contribute to the hash to avoid collisions
- Example:

```
class Person {
   String first; String middle; String last;
   int age;
}
```

- An inherent trade-off: hashing-time vs. collision-avoidance
  - Bad idea(?): Only use first name
  - Good idea(?): Only use middle initial
  - Admittedly, what-to-hash is often an unprincipled guess ☹

Hashing integers

- key space = integers
- Simple hash function:
  `h(key) = key % TableSize`
  - Client: **`f(x) = x`**
  - Library **`g(x) = x % TableSize`**
  - Fairly fast and natural
- Example:
  - TableSize = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing integers

- key space = integers
- Simple hash function:
  `h(key) = key % TableSize`
  - Client: **`f(x) = x`**
  - Library **`g(x) = x % TableSize`**
  - Fairly fast and natural
- Example:
  - TableSize = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)

0	10
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Collision-avoidance

- With "**`x % TableSize`**" the number of collisions depends on
  - the ints inserted (obviously)
  - **`TableSize`**
- Larger table-size tends to help, but not always
  - Example: 7, 18, 41, 34, 10 with **`TableSize`** = 10 and **`TableSize`** = 7
- Technique: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern, and "multiples of 61" are probably less likely than "multiples of 60"
  - Next time we'll see that one collision-handling strategy does provably better with prime table size

More on prime table size

If **`TableSize`** is 60 and…
- Lots of data items are multiples of 5, wasting 80% of table
- Lots of data items are multiples of 10, wasting 90% of table
- Lots of data items are multiples of 2, wasting 50% of table

If **`TableSize`** is 61…
- Collisions can still happen, but 5, 10, 15, 20, … will fill table
- Collisions can still happen but 10, 20, 30, 40, … will fill table
- Collisions can still happen but 2, 4, 6, 8, … will fill table

In general, if **`x`** and **`y`** are "co-prime" (means **`gcd(x,y)==1`**), then
`(a * x) % y == (b * x) % y` if and only if **`a % y == b % y`**
- So good to have a **`TableSize`** that has not common factors with any "likely pattern" **`x`**

Okay, back to the client

- If keys aren't **`int`**s, the client must convert to an **`int`**
  - Trade-off: speed and distinct keys hashing to distinct **`int`**s
- Very important example: Strings
  - Key space K = $s_0 s_1 s_2 \ldots s_{m-1}$
    - (where s_i are chars: $s_i \in [0,52]$ or $s_i \in [0,256]$ or $s_i \in [0,2^{16}]$)
  - Some choices: Which avoid collisions best?

1. h(K) = s_0 % TableSize
2. h(K) = $\left(\sum_{i=0}^{m-1} s_i\right)$ % TableSize
3. h(K) = $\left(\sum_{i=0}^{k-1} s_i \cdot 37^i\right)$ % TableSize

Specializing hash functions

How might you hash differently if all your strings were web addresses (URLs)?

Combining hash functions

A few rules of thumb / tricks:

1. Use all 32 bits (careful, that includes negative numbers)
2. Use different overlapping bits for different parts of the hash
   - This is why a factor of 37^i works better than 256^i
   - Example: “abcde” and “ebcda”
3. When smashing two hashes into one hash, use bitwise-xor
   - bitwise-and produces too many 0 bits
   - bitwise-or produces too many 1 bits
4. Rely on expertise of others; consult books and other resources
5. If keys are known ahead of time, choose a *perfect hash*