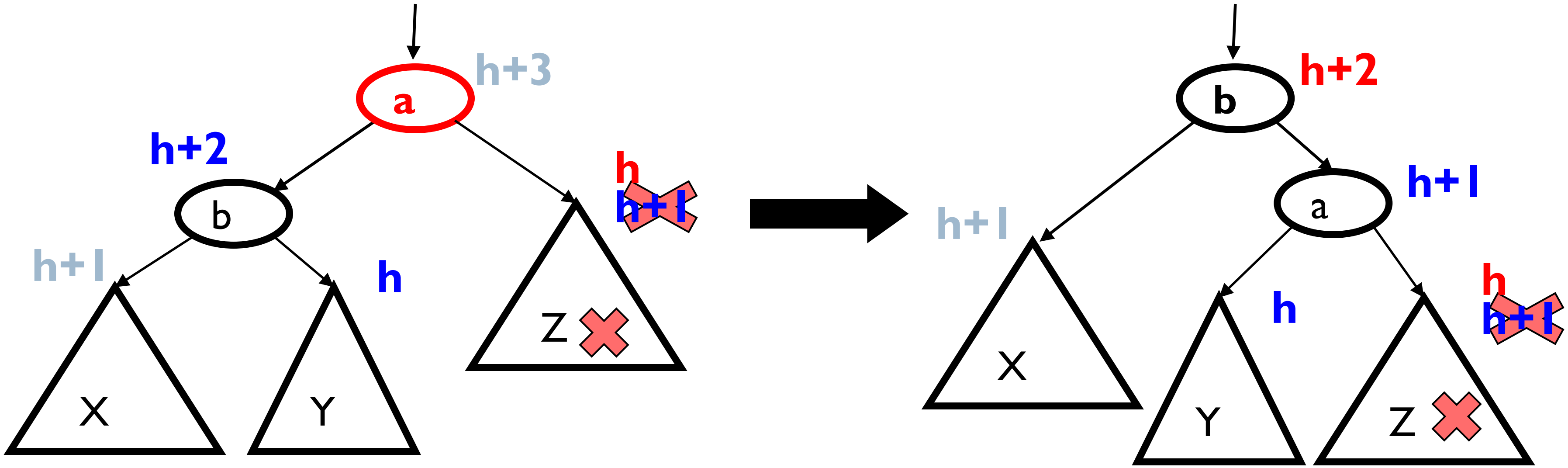
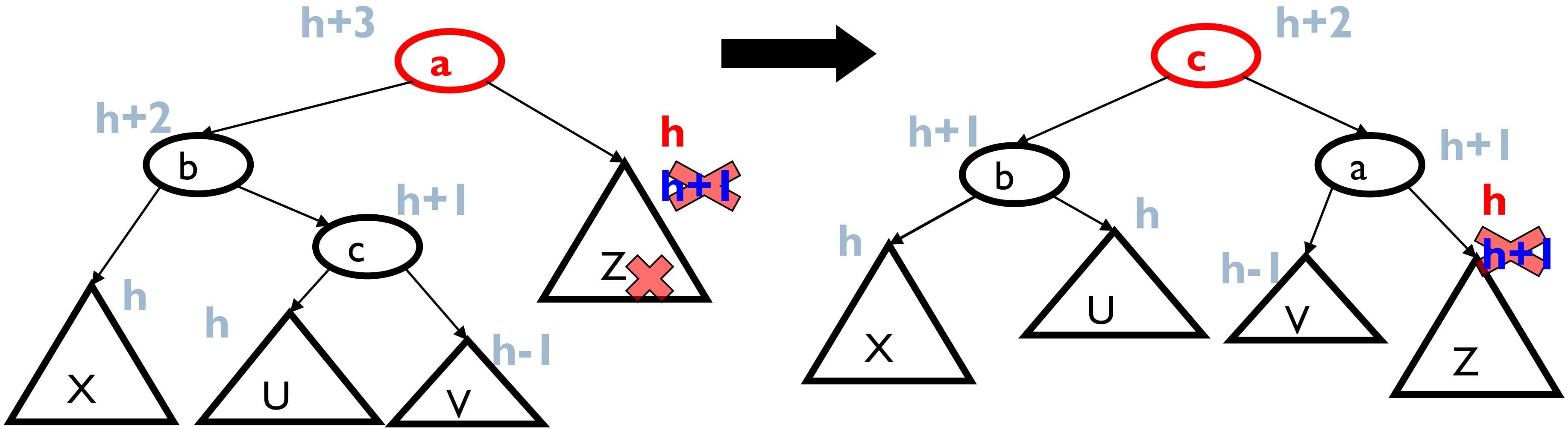
AVL Deletion:
Case #1: Left-left due to right deletion



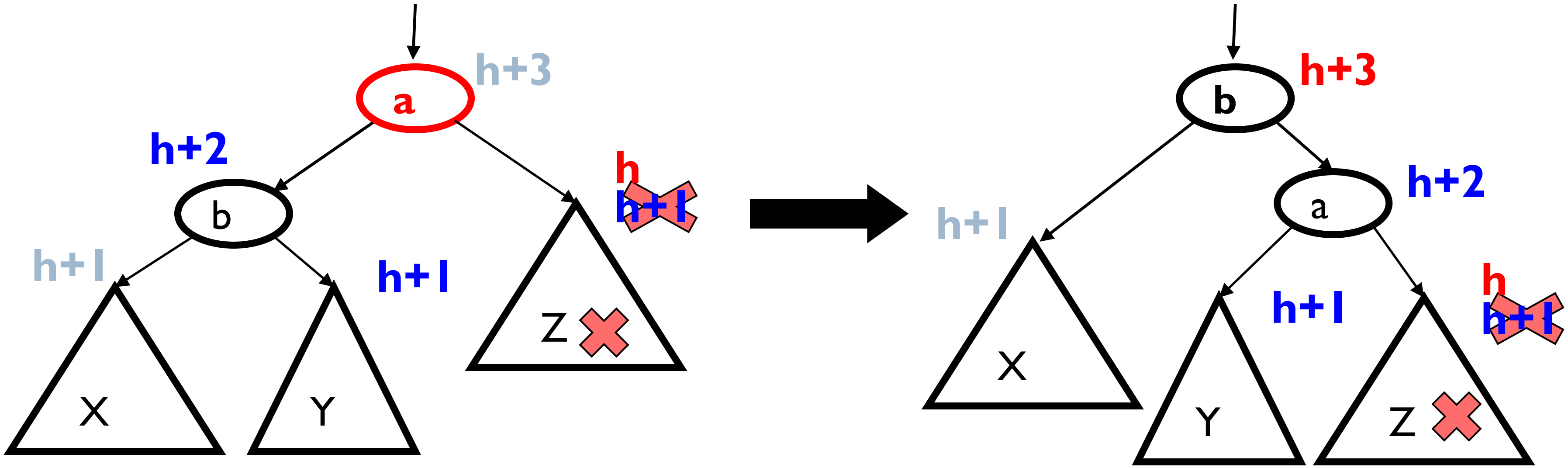
- Same single rotation as when an insert in the left-left grandchild caused imbalance due to X becoming taller
- But here the "height" at the top decreases, so more rebalancing farther up the tree might still be necessary

AVL Deletion:
Case #2: Left-right due to right deletion



- Same double rotation when an insert in the left-right grandchild caused imbalance due to c becoming taller
- But here the “height” at the top decreases, so more rebalancing farther up the tree might still be necessary

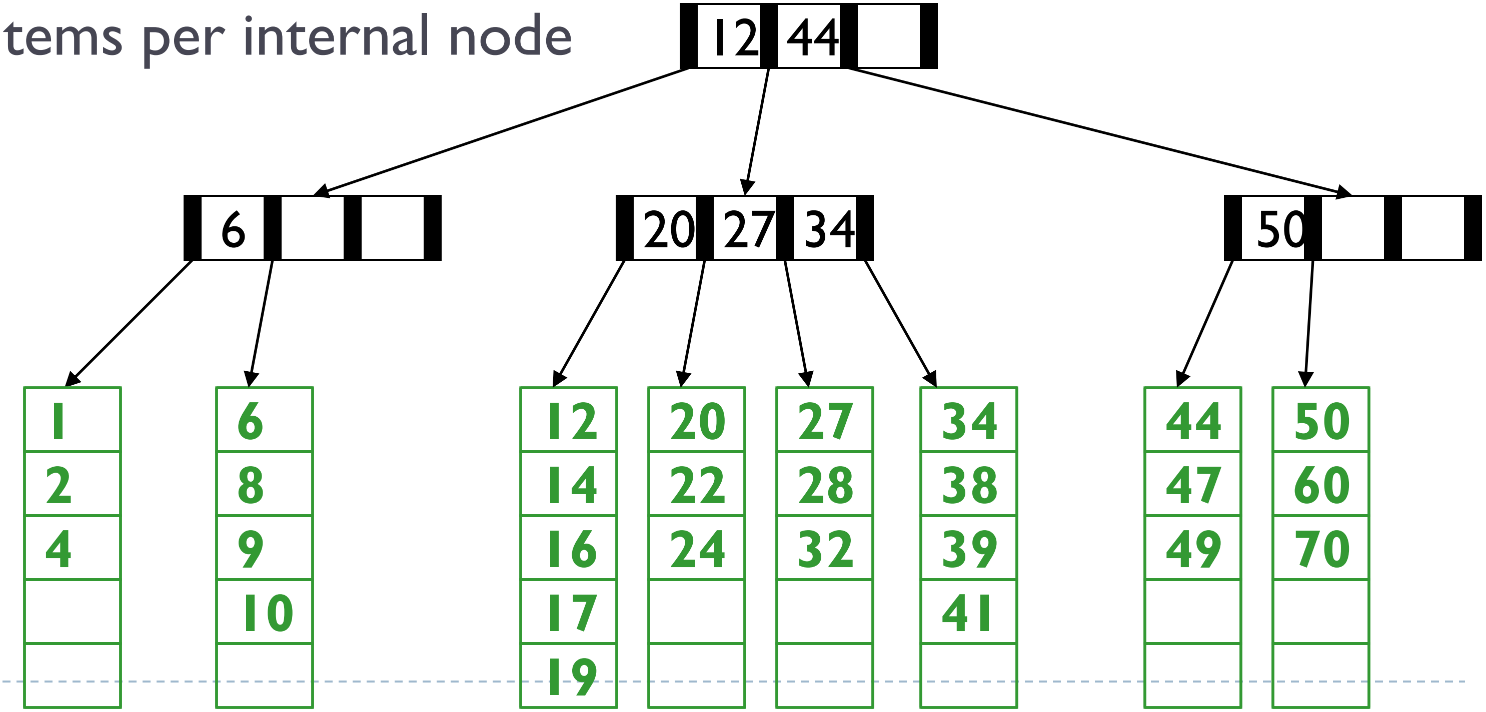
AVL Deletion:
Case #3: Case 1 revisited



- What if both children have same height (h+1)?
- Do same as case 1; single rotation
- Why can't we do the double rotation from case 2?

B-Trees

- Smaller keys on left, larger on right
- All data in leaves
- Need to decide:
  - # of items per leaf
  - # of items per internal node



B-Tree Operations

Insertion

1. Traverse from the root to the proper leaf. Insert the data in its leaf in sorted order
2. If the leaf now has L+1 items, *overflow!*
   - Split the leaf into two leaves:
     - Original leaf with $\lceil (L+1)/2 \rceil$ items
     - New leaf with $\lfloor (L+1)/2 \rfloor$ items
   - Attach the new child to the parent
     - Adding new key to parent in sorted order
3. If an internal node has M+1 children, *overflow!*
   - Split the node into two nodes
     - Original node with $\lceil (M+1)/2 \rceil$ children
     - New node with $\lfloor (M+1)/2 \rfloor$ children
   - Attach the new child to the parent
     - Adding new key to parent in sorted order

Splitting at a node (step 3) could make the parent overflow too

- So repeat step 3 up the tree until a node doesn't overflow
- If the root overflows, make a new root with two children
  - This is the only case that increases the tree height

Deletion

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!*
   - 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

- Somewhat complex; we won't go into details…

Aside: Limitations of B-Trees in Java

Whole point of B-Trees is to minimize disk accesses

It is worth knowing enough about "how Java works" to understand why B-Trees in Java aren't what we want

- Assuming our goal is efficient number of disk accesses
- Java has many advantages, but it wasn't designed for this

The problem is extra *levels of indirection*…

One approach

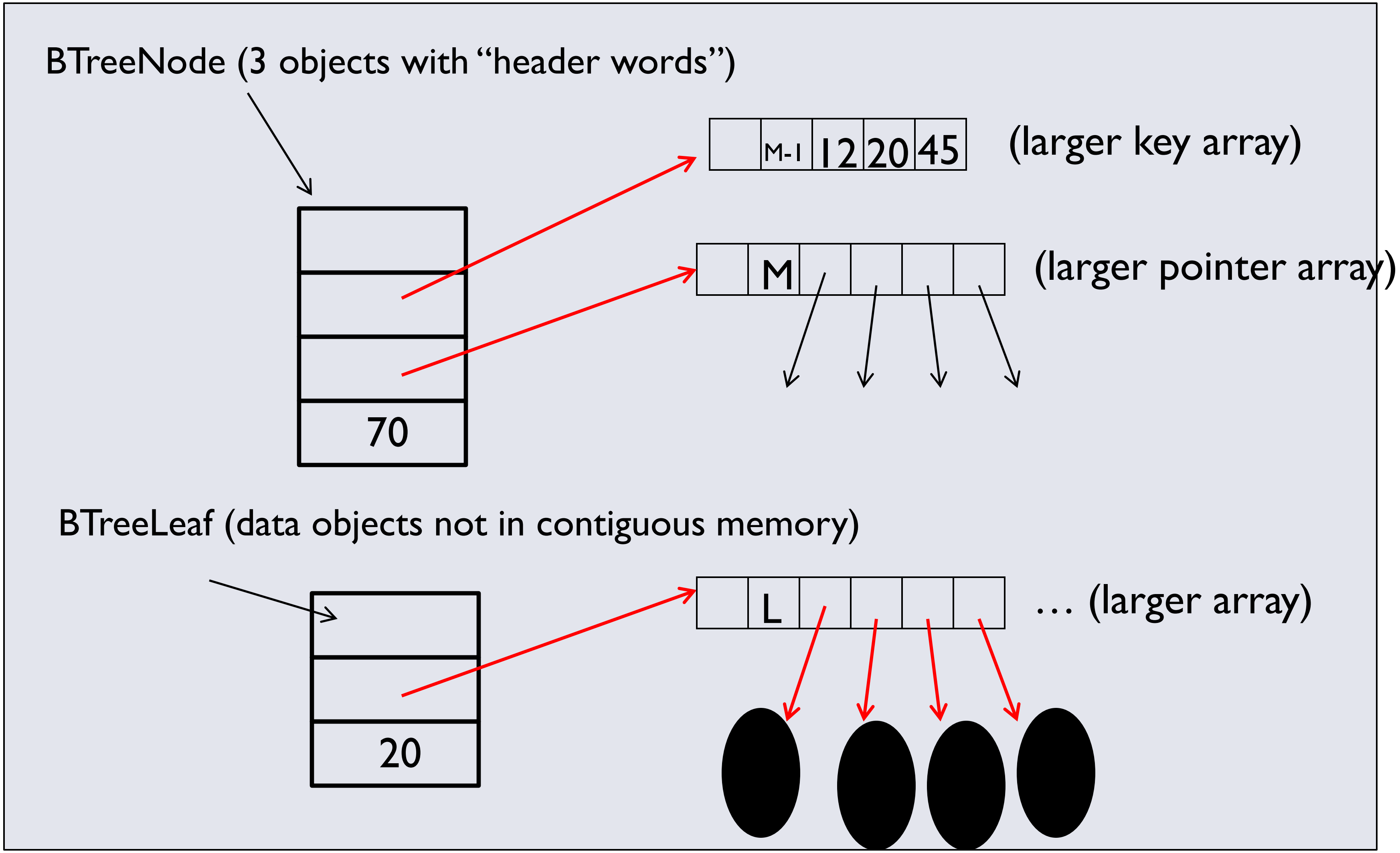
Say we int keys, and some data E

```
interface Keyed<E> {
   int key(E);
}
class BTreeNode<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;
   …
}
```

The problem: how Java stores stuff in memory

What that looks like

All the red references indicate unnecessary indirection


BTreeNode (3 objects with "header words")

(larger key array)

(larger pointer array)

BTreeLeaf (data objects not in contiguous memory)

… (larger array)

The moral

- The whole idea behind B trees was to keep related data in contiguous memory
- But that's "the best you can do" in Java
  - Java's advantage is generic, reusable code
- C# may have better support for "flattening objects into arrays"
  - C and C++ definitely do
- Levels of indirection matter!

Picking a hash function

- 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: $s_i \in [0,51]$ or $s_i \in [0,255]$ or $s_i \in [0,2^{16}-1]$
  - Some choices: Which avoid collisions best?

1. h(K) = s_0 % TableSize

   Anything w/ same first letter

2. $$h(K) = \left(\sum_{i=0}^{m-1} s_i \right) \% \text{ TableSize}$$

   Any rearrangement of letters

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

   Hmm… not so clear

What causes collisions for each?

Java-esque String Hash

- Java characters in Unicode format; 2^{16} bits

$$h = s[0]*31^{n-1} + s[1]*31^{n-2} + \cdots + s[n-1]$$

- So this would require n-2 + n-3 + n-4 +… multiplications to compute, right?
- Can compute efficiently via a trick called Horner's Rule:
  - Idea: Avoid expensive computation of 31^k
  - Say n=4
  - h=((s[0]*31+s[1])*31+s[2])*31+s[3]
- Under what circumstances could this hash function prove poor?

Hash functions

A few rules of thumb / tricks:

1. Use all 32 bits (careful, that includes negative numbers)
2. When smashing two hashes into one hash, use bitwise-xor
   - Problem with Bitwise AND?
     - Produces too many 0 bits
   - Problem with Bitwise OR?
     - Produces too many 1 bits
3. Rely on expertise of others; consult books and other resources
4. If keys are known ahead of time, choose a *perfect hash*