Faculty of Information Technology, Monash University

FIT2004: Algorithms and Data Structures

Week 5: Efficient Lookup Structures

These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Things to note/remember

- Assignment 2 due 1 May 2020 - 23:55:00

Recommended Reading

- Unit Notes – chapters 7 and 8

Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables

Lookup Table

- A lookup table allows inserting, searching and deleting values by their keys.
- The idea of a lookup table is very general and important in information processing systems.
- The database that Monash University maintains on students is an example of a table. This table might contain information about:
  - Student ID
  - Authcate
  - First name
  - Last name
  - Course(s) enrolled

Lookup Table

- Elements of a table, in some cases, contain a key plus some other attributes or data
- The key might be a number or a string (e.g., Student ID or authcate)
- It is something unique that unambiguously identifies an element
- Elements can be looked up (or searched) using this key

Sorting based lookup

Keep the elements sorted on their keys in an array (e.g., sort by student ID)

Searching:

- O(log N)
  - use Binary search to find the key – O(log N)

Insertion:

- O(N)
  - Use Binary search to find the sorted location of new element – O(log N)
  - Insert the new element and shift all larger elements toward right – O(N)

Deletion:

- O(N)
  - Search the key – O(log N)
  - Delete the key – O(1)
  - Shift all the larger elements to left – O(N)

Is it possible to do better?

Yes! Hash Tables and Balanced Binary Search trees (to name a few)

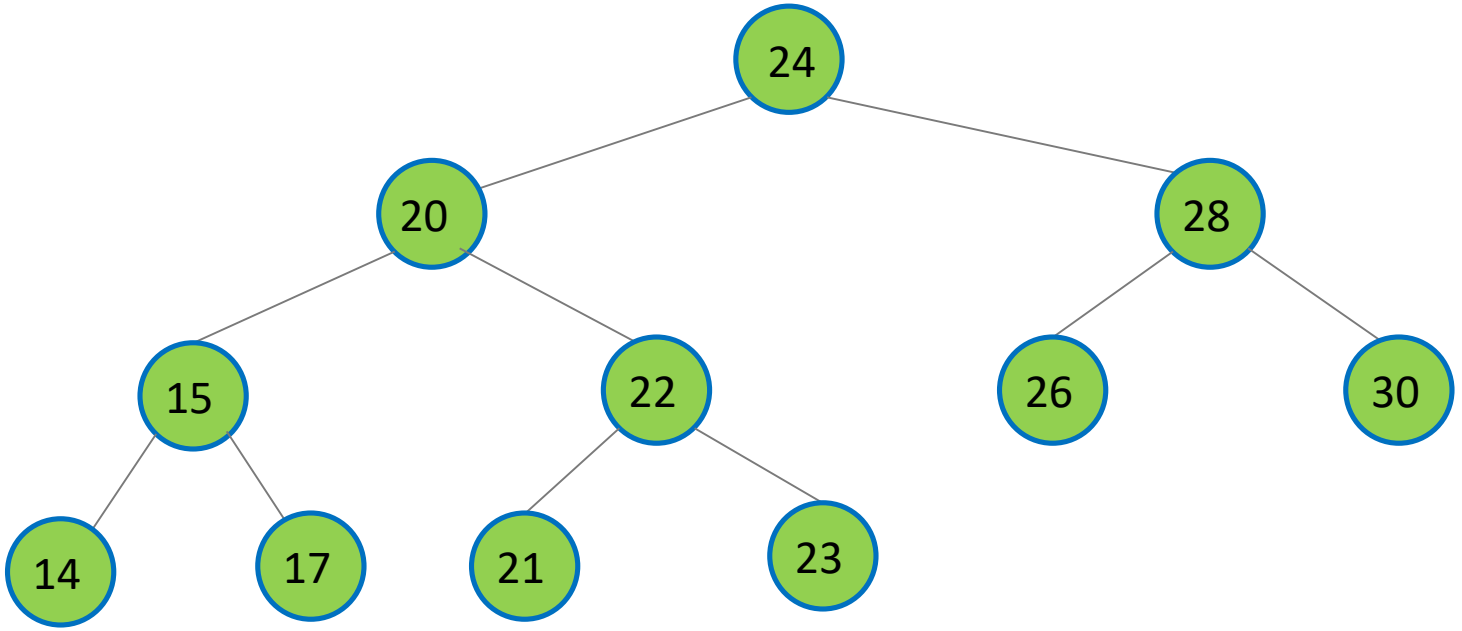
Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables

Binary Search Tree (BST)

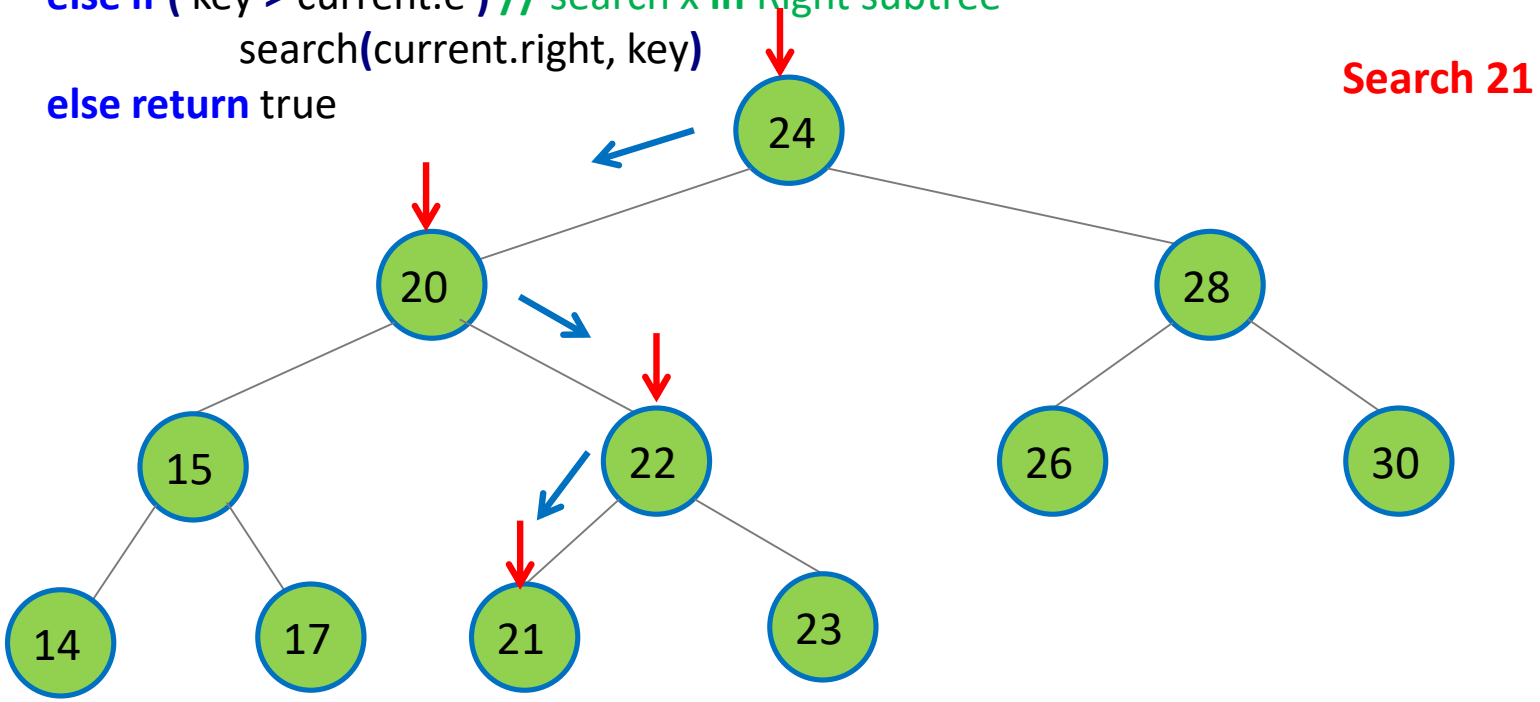
- The empty tree is a BST
- If the tree is not empty
  1. the elements in the left subtree are LESS THAN the element in the root
  2. the elements in the right subtree are GREATER THAN the element in the root
  3. the left subtree is a BST
  4. the right subtree is a BST

Note! Don't forget last two conditions!



Searching a key in BST

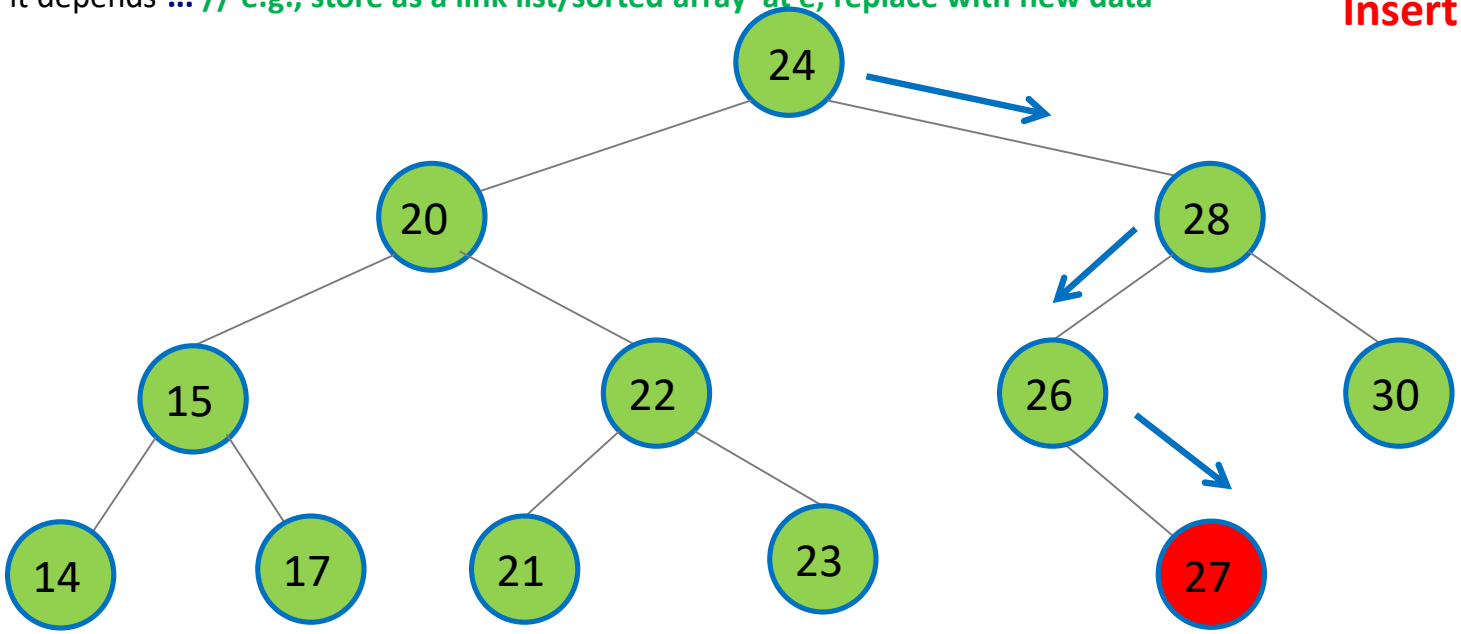
```
// BST implemented here as a tree data structure
def search(current, key)
        if (current == None)
            return false // not present!
        else if ( key < current.e ) // search x in Left subtree
                search(current.left, key)
        else if ( key > current.e ) // search x in Right subtree
                search(current.right, key)
        else return true
```



Insert a key x in BST

```
// BST implemented here as a tree data structure
// T = fork(e, L, R)
function insert(current, x)
        if (current == None) // Insert here as leaf node
                set root to x
        else if ( x < current.e ) // Traverse and insert ...
                insert( current.left, x); // along the Left subtree
        else if (x > current.e ) // Traverse and insert ...
                insert( current.right, x) // along the Right subtree
        else // x == e
                ... it depends ... // e.g., store as a link list/sorted array  at e, replace with new data
        return
```

Insert 27



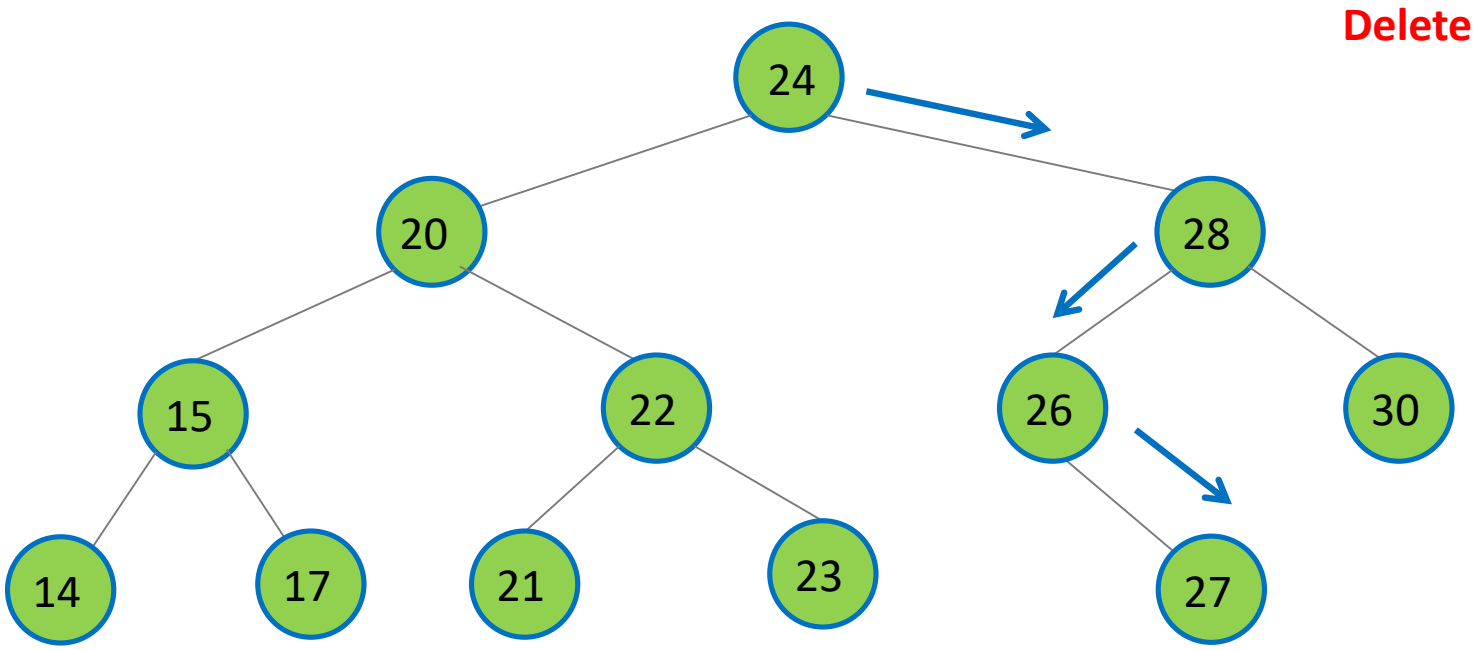
Delete a key from BST

First lookup key in BST

If the key node has no children // Case 1

delete the key node

set subtree to nil



Delete a key from BST

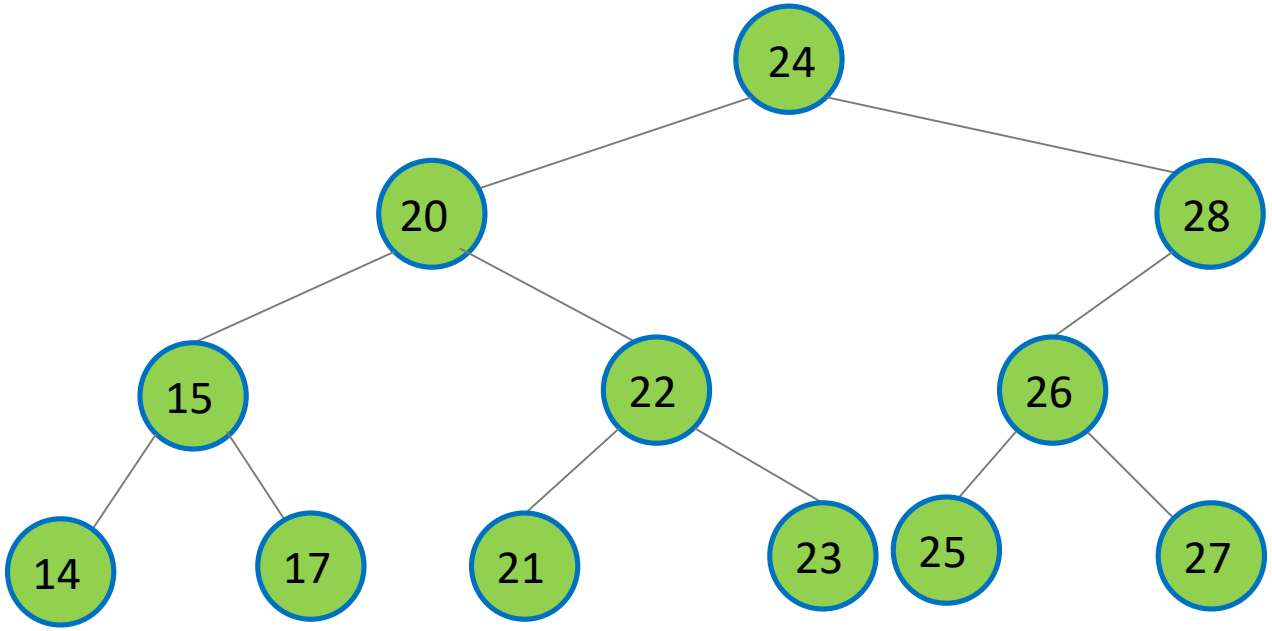
First lookup key in BST

If the key node has one child // Case 2

delete the key node

replace the key node with its child

Delete 28



Delete a key from BST

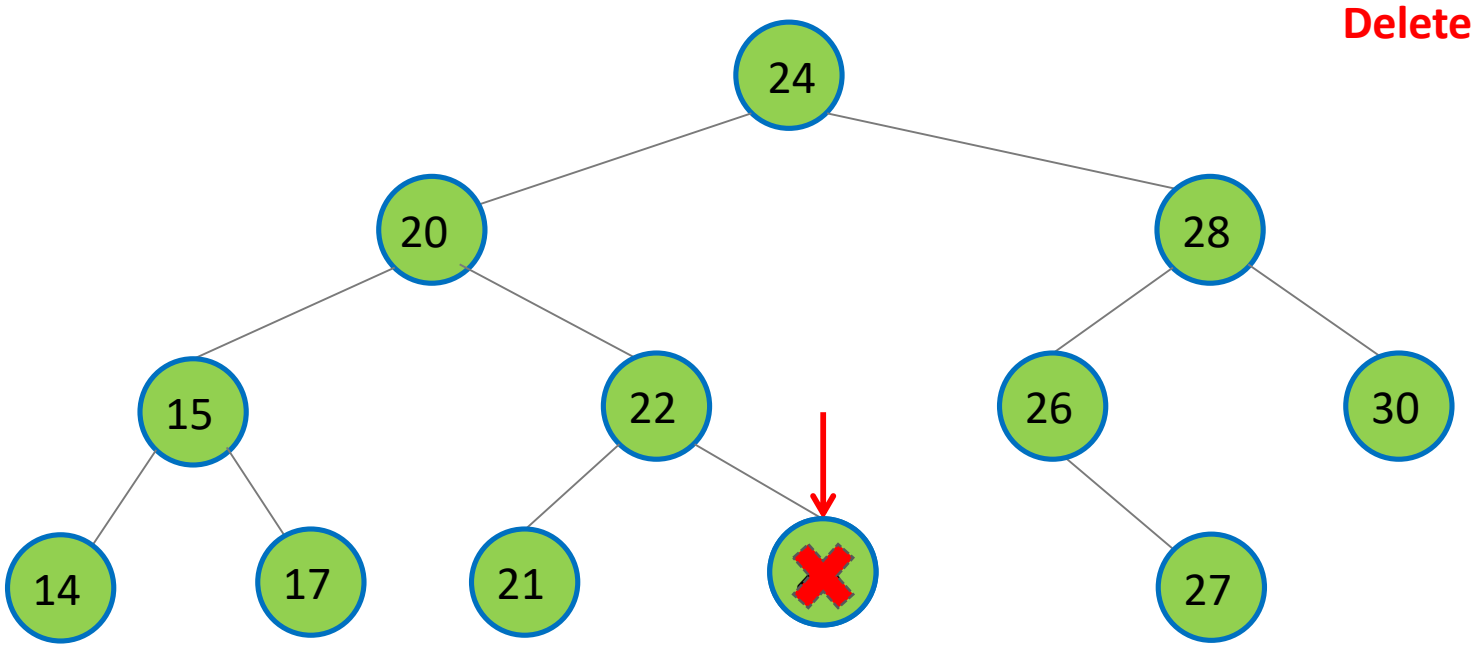
First lookup key in BST

If the key node has two children // Case 3

Find successor (or predecessor)

Replace key node value with the value of successor (or predecessor)

Delete successor (or predecessor)



Delete a key from BST

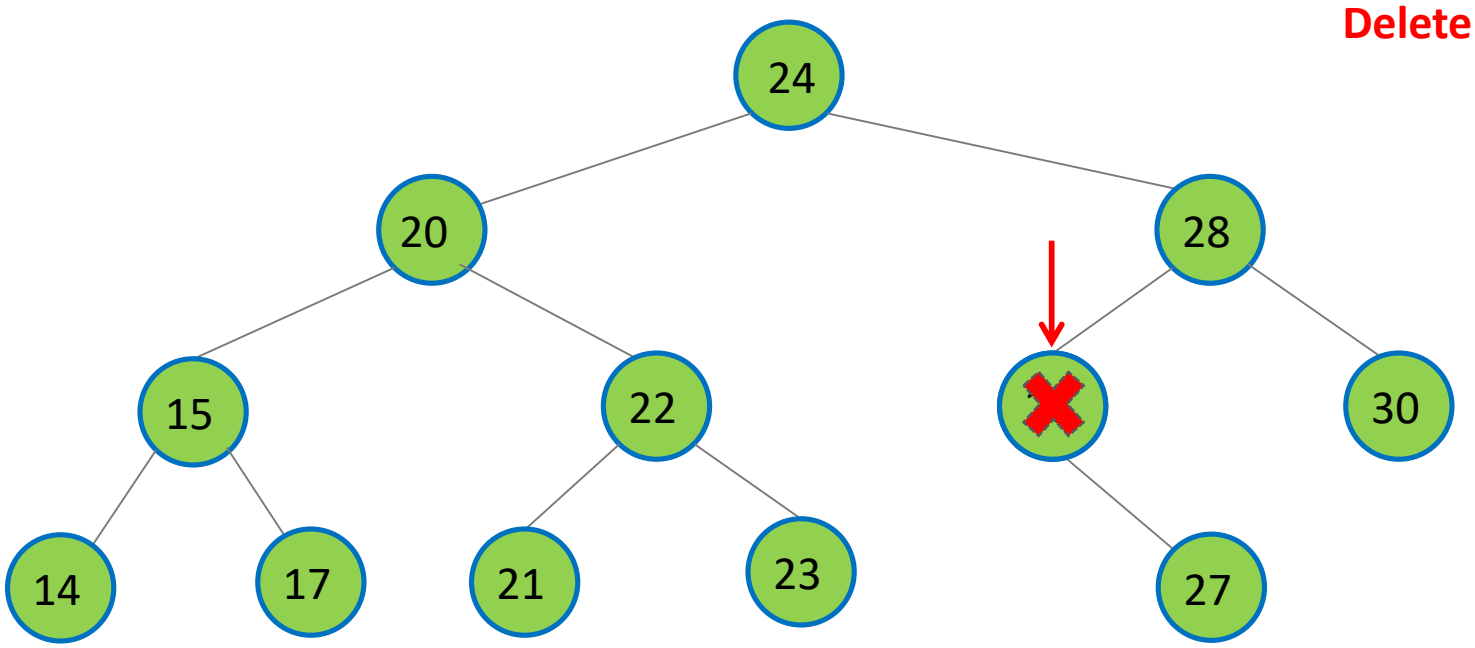
First lookup key in BST

If the key node has two children // Case 3

Find successor (or predecessor)

Replace key node value with the value of successor (or predecessor)

Delete successor (or predecessor)

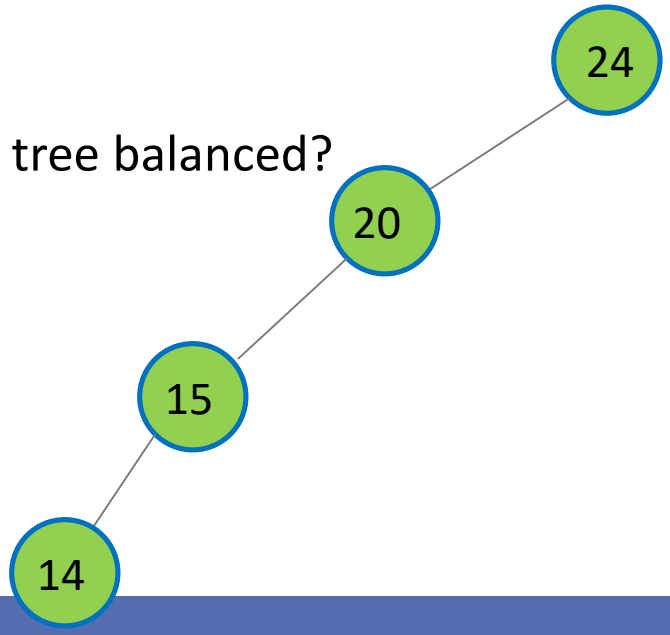


Worst-case of BST

- A BST is not a balanced tree and, in worst case, may degenerate to a linked list
  - E.g., when elements are inserted in sorted order (ascending or descending) – insert 24, 20, 15, 14.

- Worst-case time complexity
  - Insert: O(N)
  - Delete: O(N)
  - Search: O(N)

- Can we improve the performance by keeping the tree balanced?



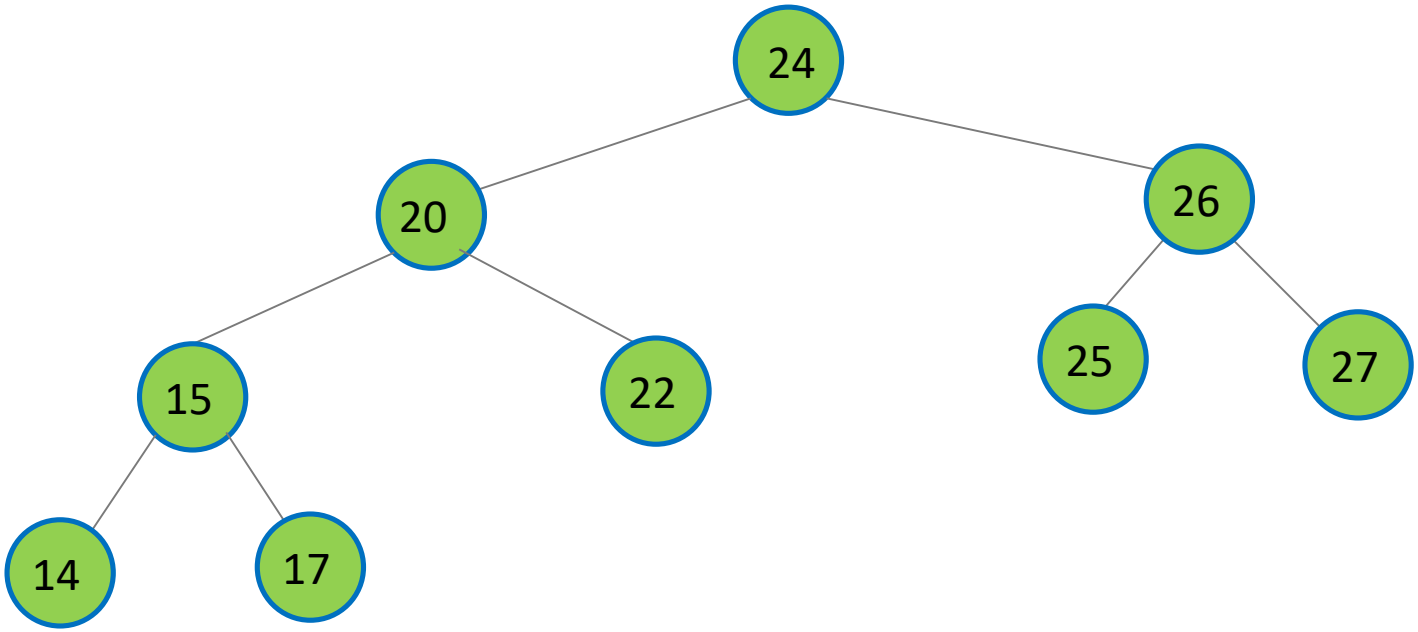
Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
   A. Introduction
   B. Balancing AVL tree
   C. Complexity Analysis
4. Hash tables

AVL Trees: Introduction

- Adelson-Velskii Landis (AVL) tree is a height-balanced BST
- The heights of left and right subtrees of **every** node differ by at most one
  - If at any time they differ by more than one, rebalancing is done to restore this property.

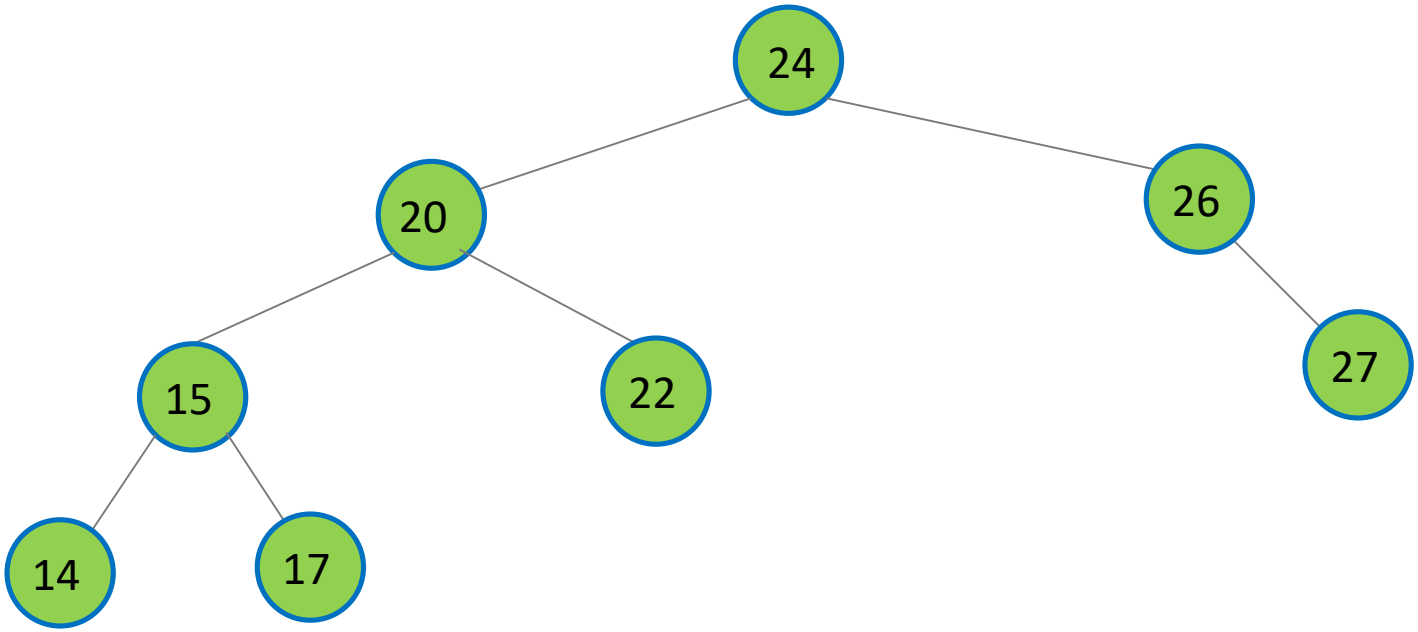
- Is the following tree balanced according to the above definition?



AVL Trees: Introduction

- Adelson-Velskii Landis (AVL) tree is a height-balanced BST
- The heights of left and right subtrees of **every** node differ by at most one
  - If at any time they differ by more than one, rebalancing is done to restore this property.

- Is it still balanced after deleting 25?

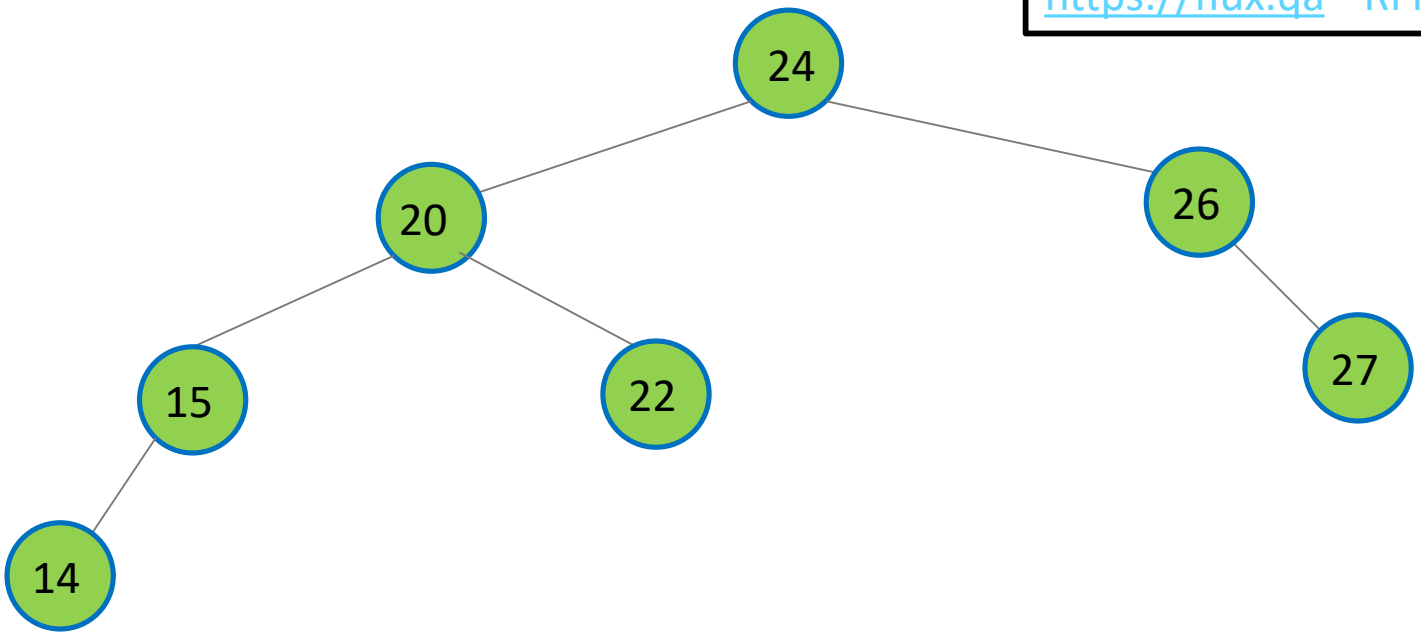


AVL Trees: Introduction

- Adelson-Velskii Landis (AVL) tree is a height-balanced BST
- The heights of left and right subtrees of **every** node differ by at most one
  - If at any time they differ by more than one, rebalancing is done to restore this property.

- Is it still balanced after deleting 17?

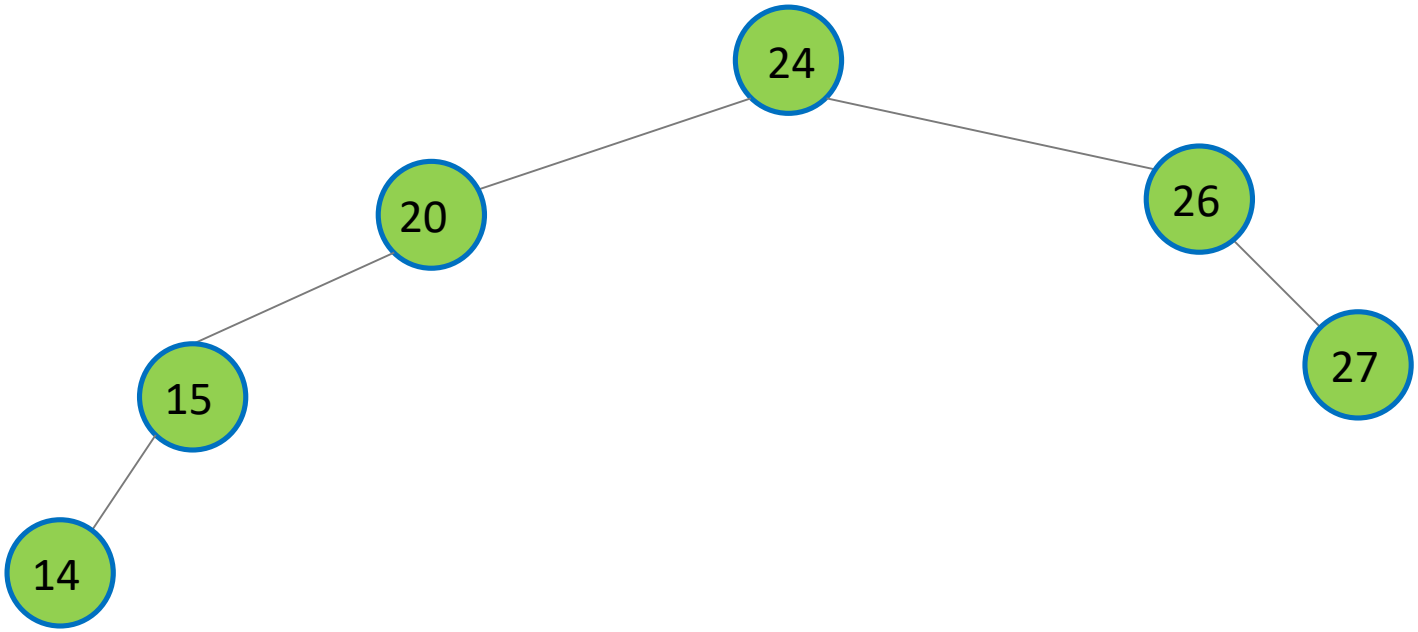
Quiz time!
https://flux.qa - RFIBMB



AVL Trees: Introduction

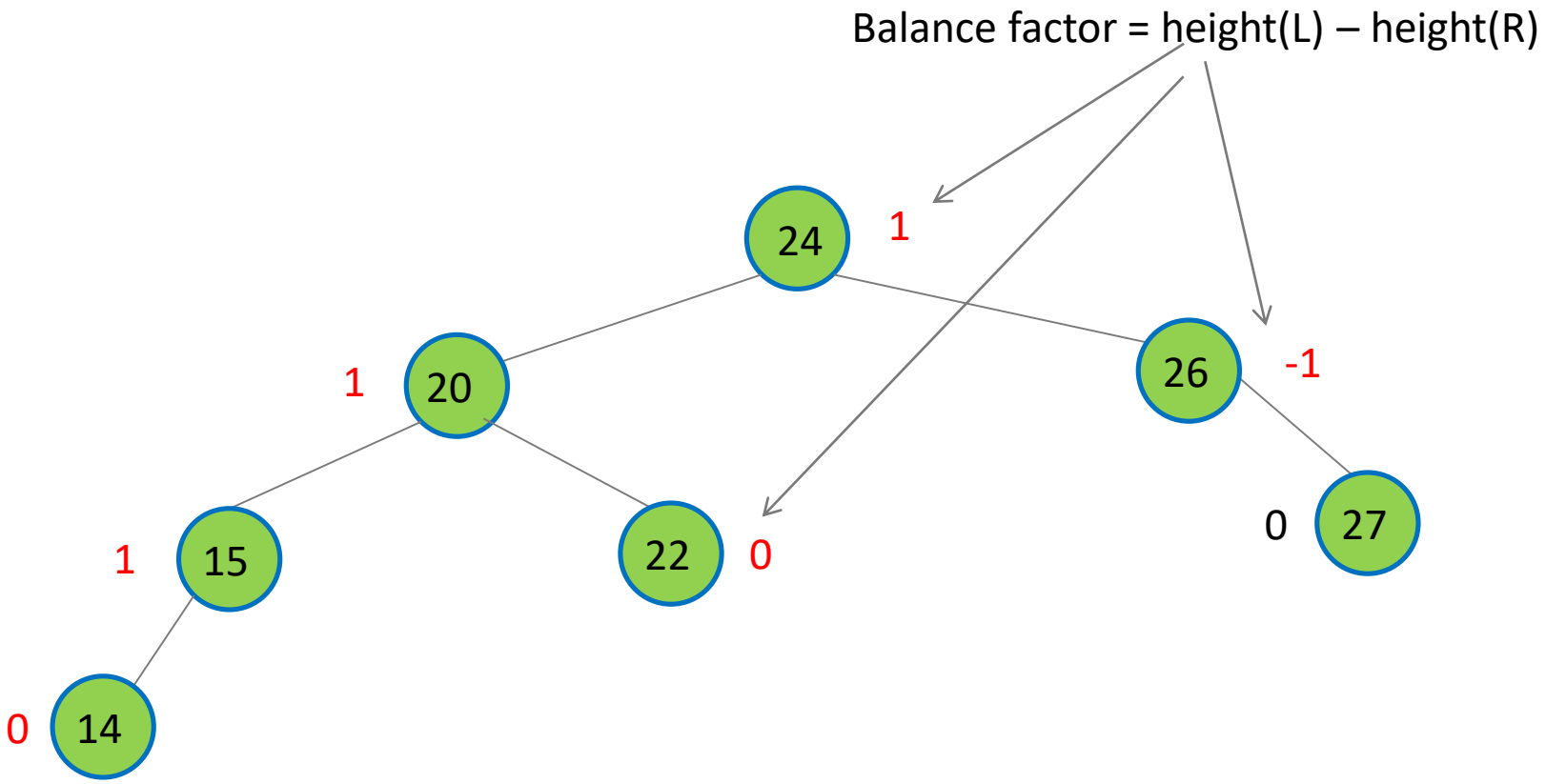
- Adelson-Velskii Landis (AVL) tree is a height-balanced BST
- The heights of left and right subtrees of **every** node differ by at most one
  - If at any time they differ by more than one, rebalancing is done to restore this property.

- Is it still balanced after deleting 22?



Defining AVL Tree

- T is an AVL Tree if T is a binary search tree, and . . .
- Every node of T has a balance_factor 1,0, or -1

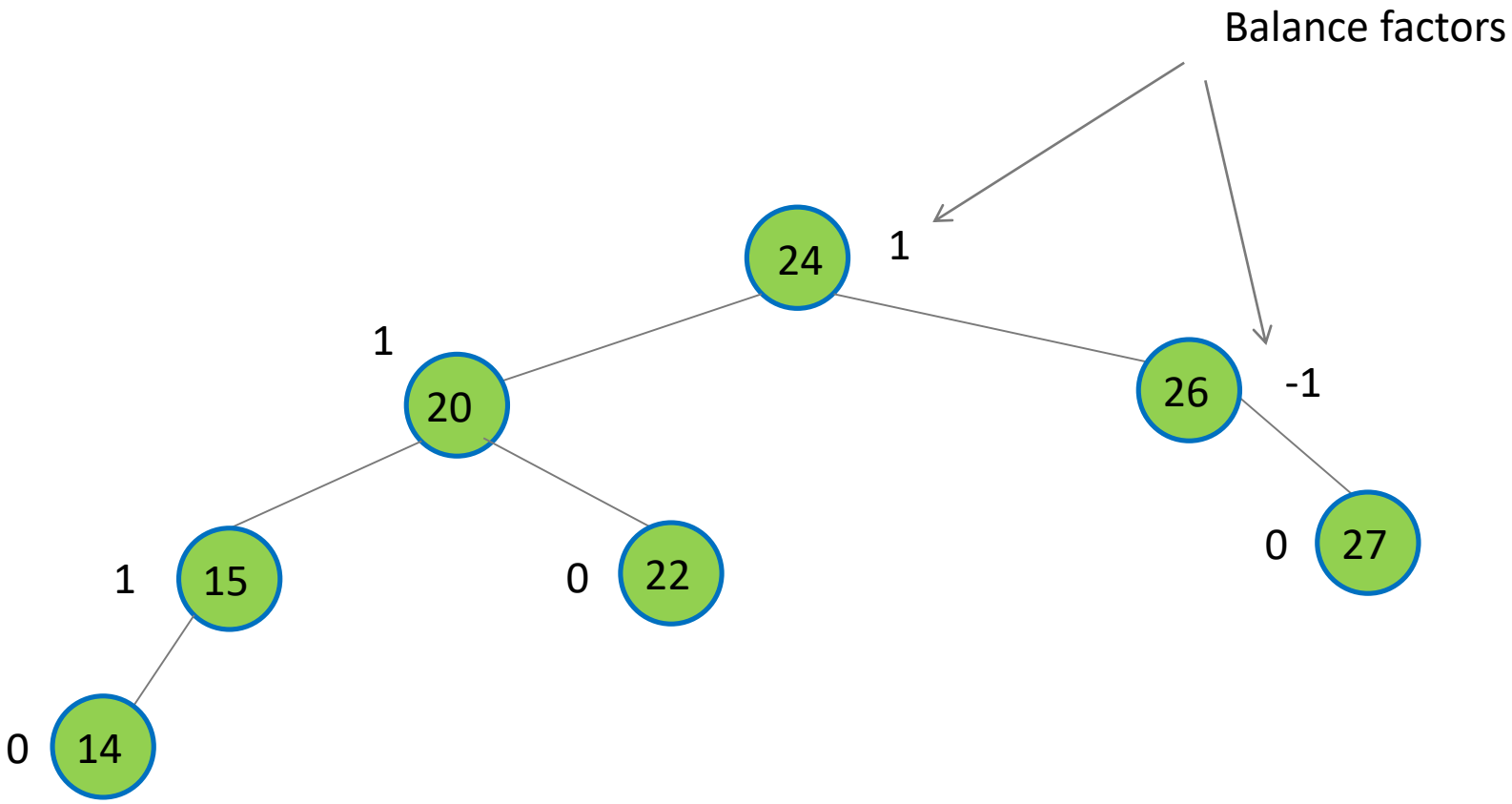


Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
   A. Introduction
   B. Balancing AVL tree
   C. Complexity Analysis
4. Hash tables

Balancing AVL Tree after insertion/deletion

- The tree becomes unbalanced after deleting 22.



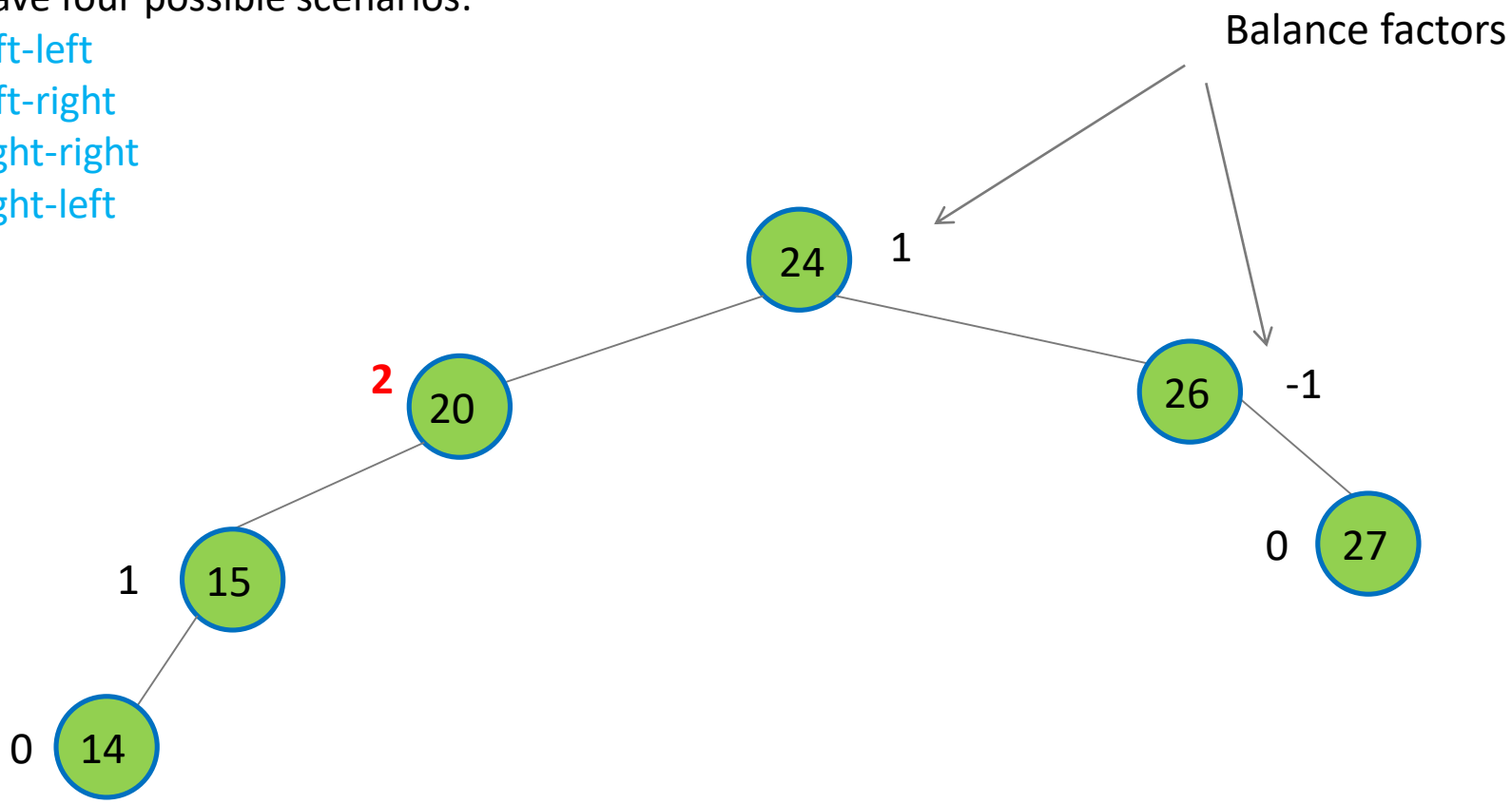
Balancing AVL Tree after insertion/deletion

- The tree becomes unbalanced after deleting 22.
- The tree may also become unbalanced after insertion.

How to balance it after deletion/insertion?

We have four possible scenarios:

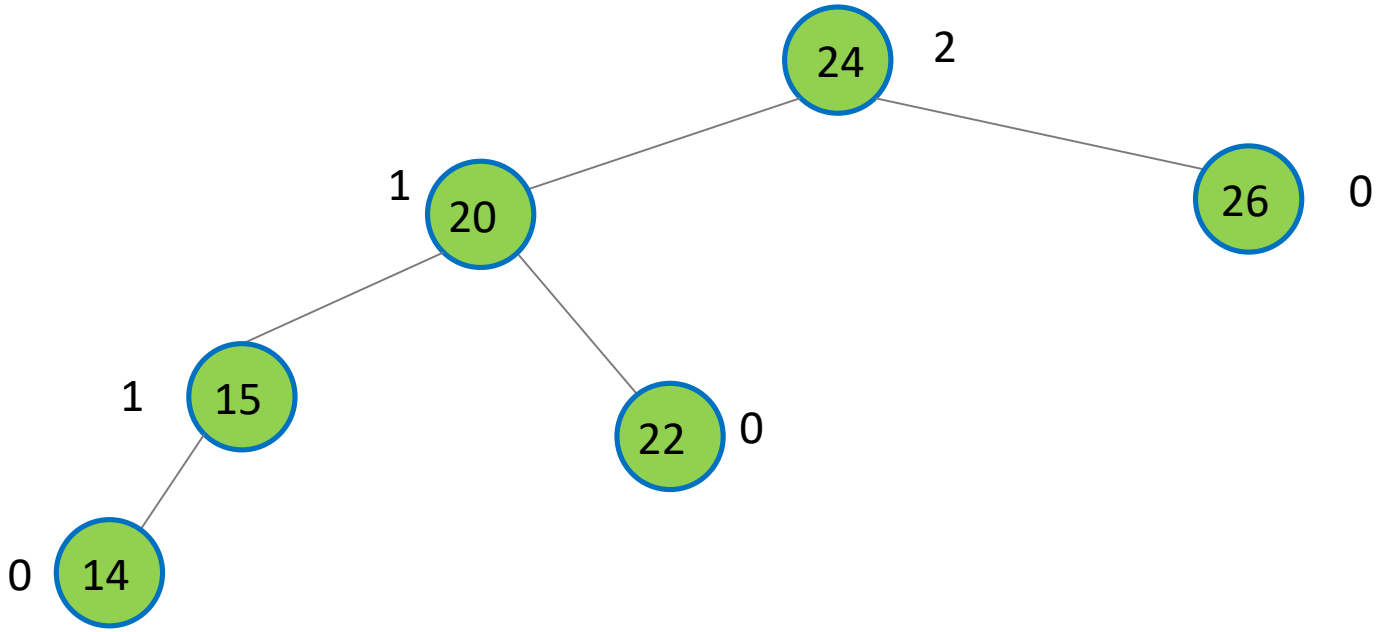
- Left-left
- Left-right
- Right-right
- Right-left



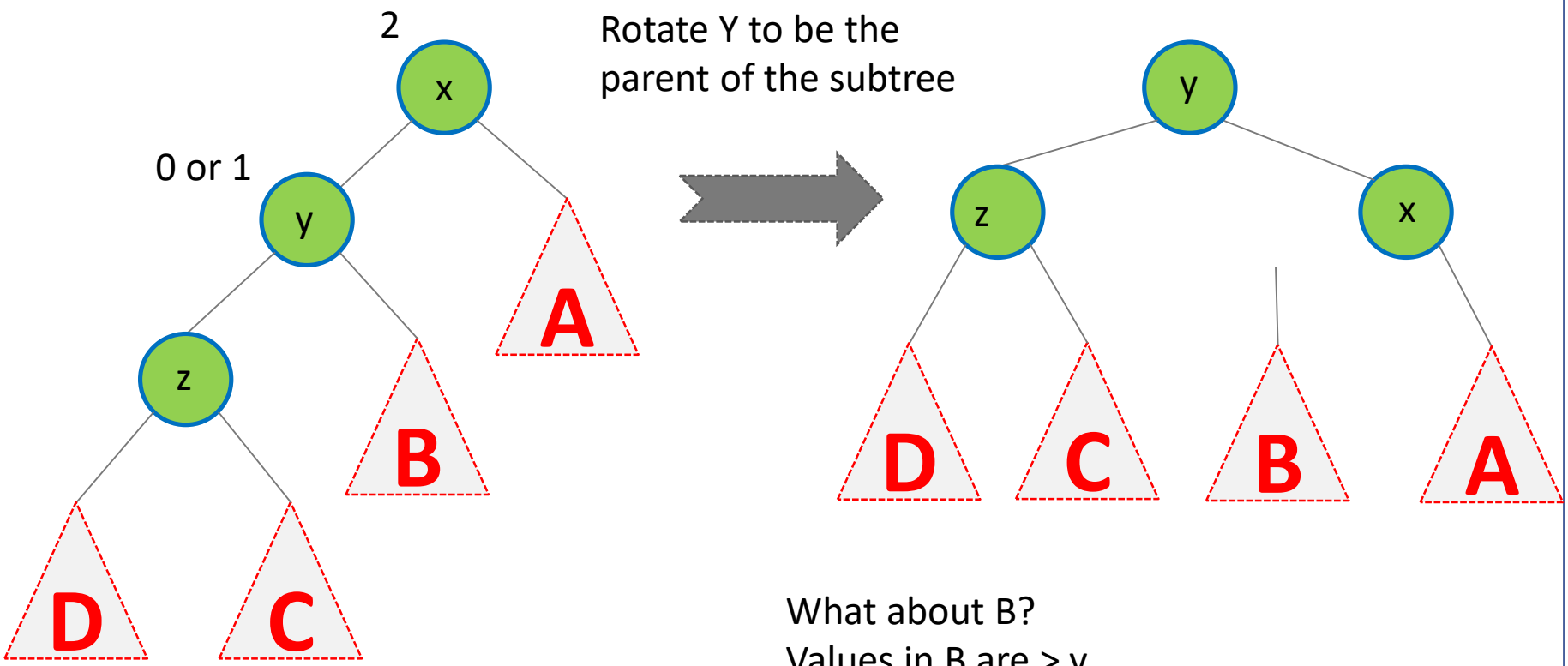
Left-Left Case

A left-left case occurs when

- A node has a balance factor +2; and
- Its left child has a balance factor 0 or more

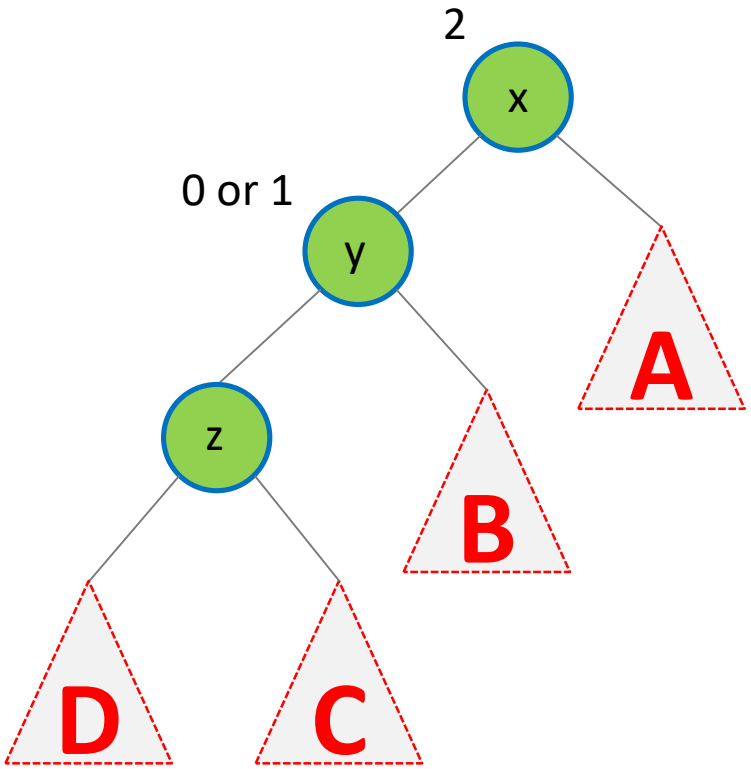


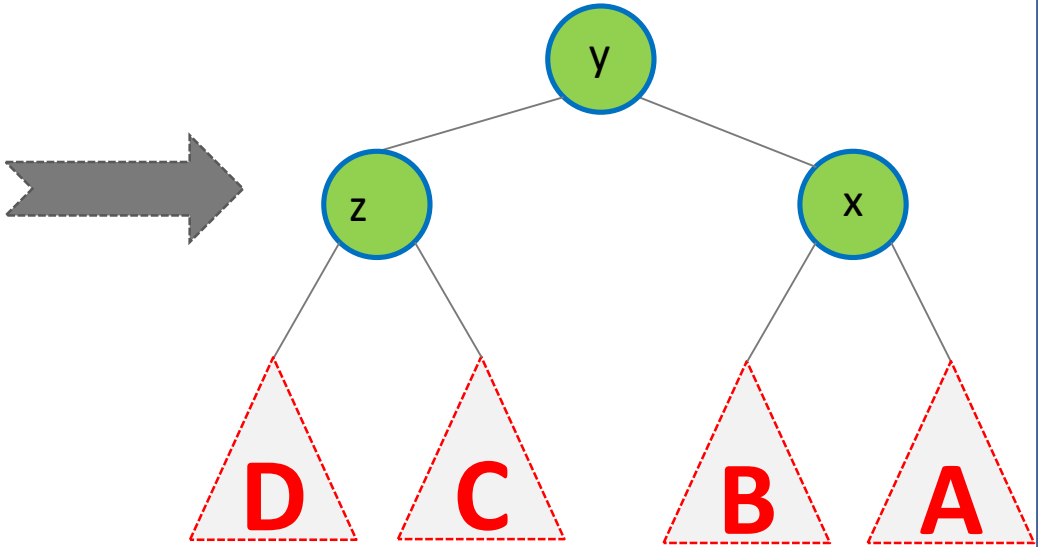
Handling Left-left case



What about B?
Values in B are > y
Values in B are < x
B can be the left child of x!

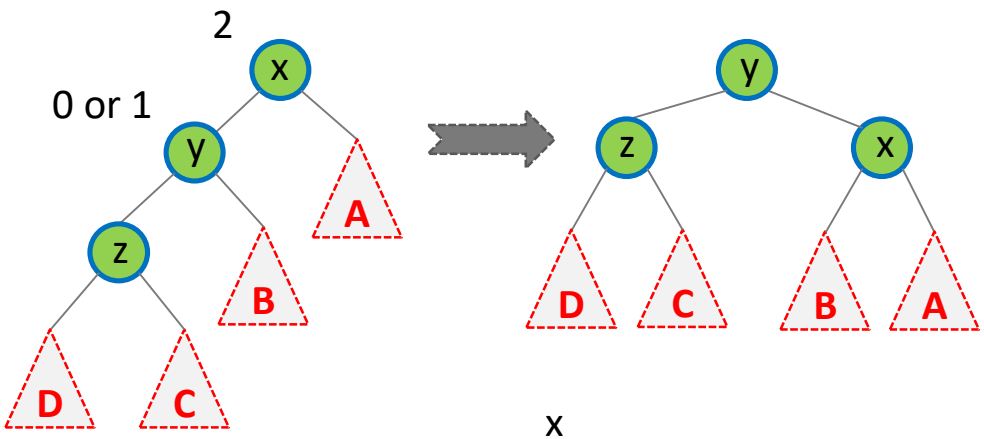
Handling Left-left case

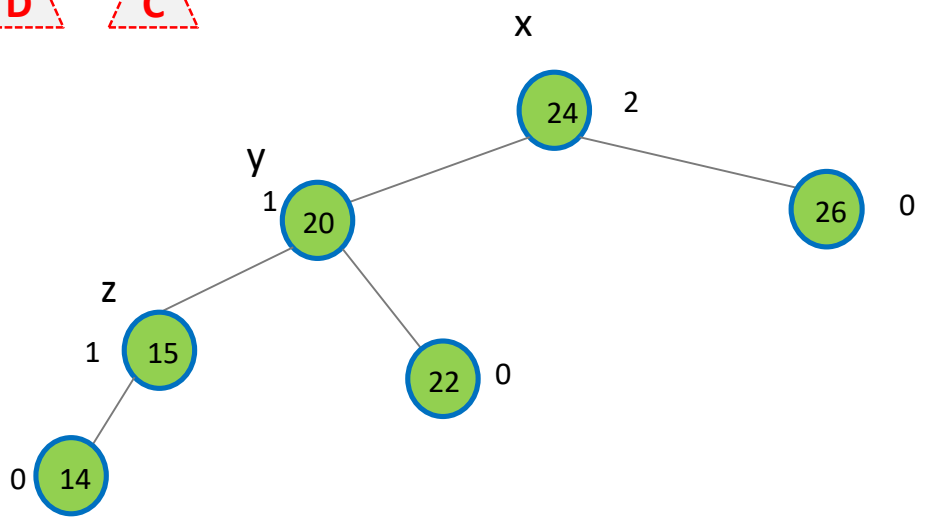




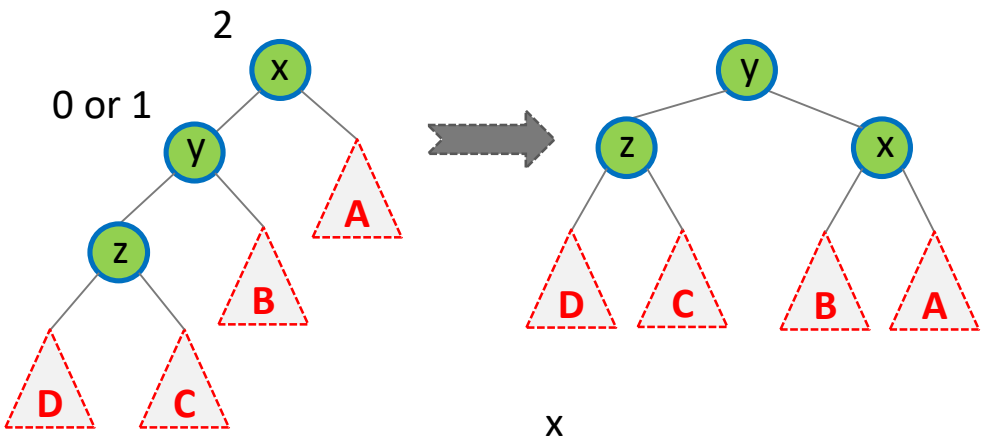
What about B?
Values in B are > y
Values in B are < x
B can be the left child of x!

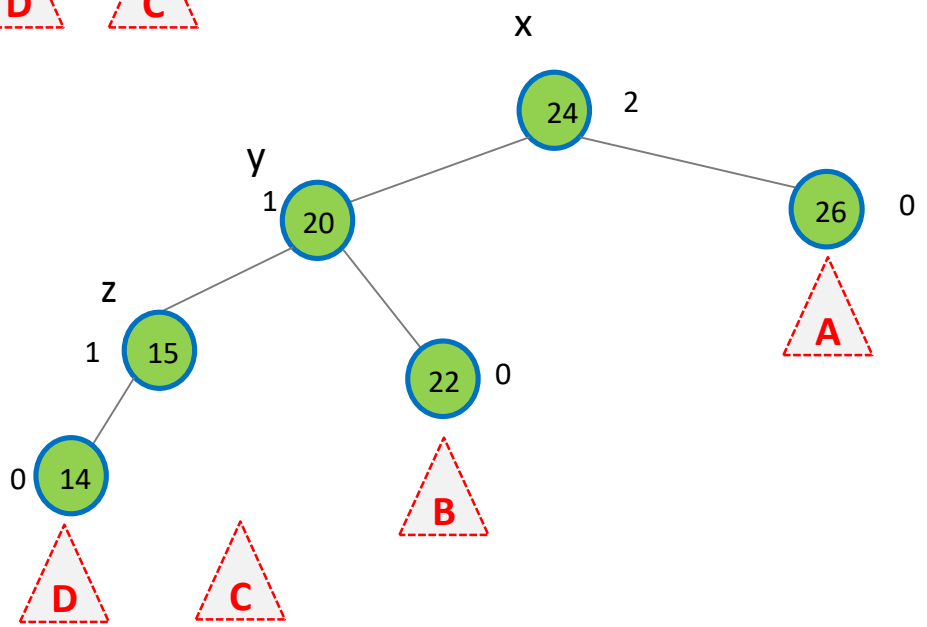
Example: Left-left case



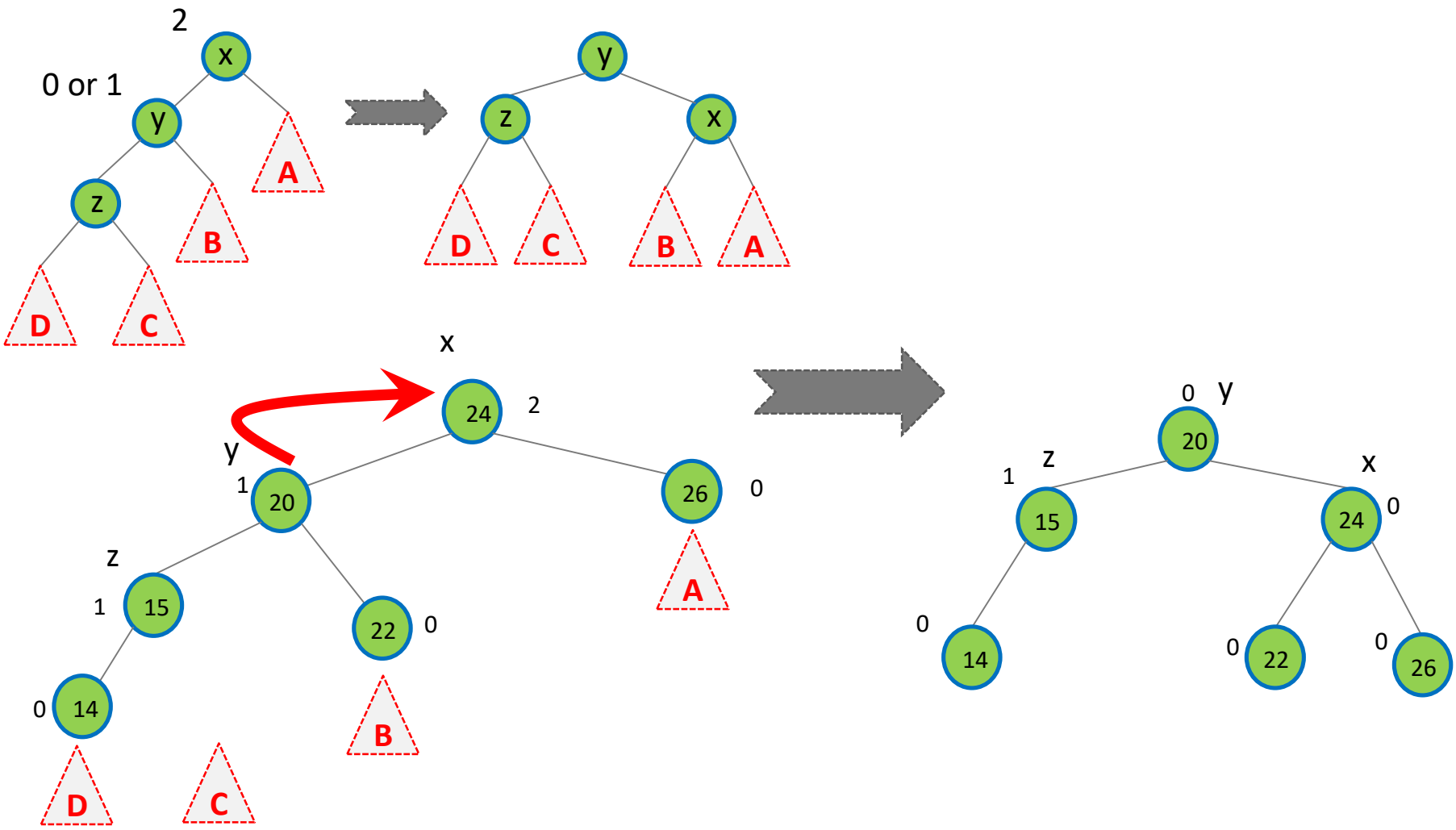


Example: Left-left case

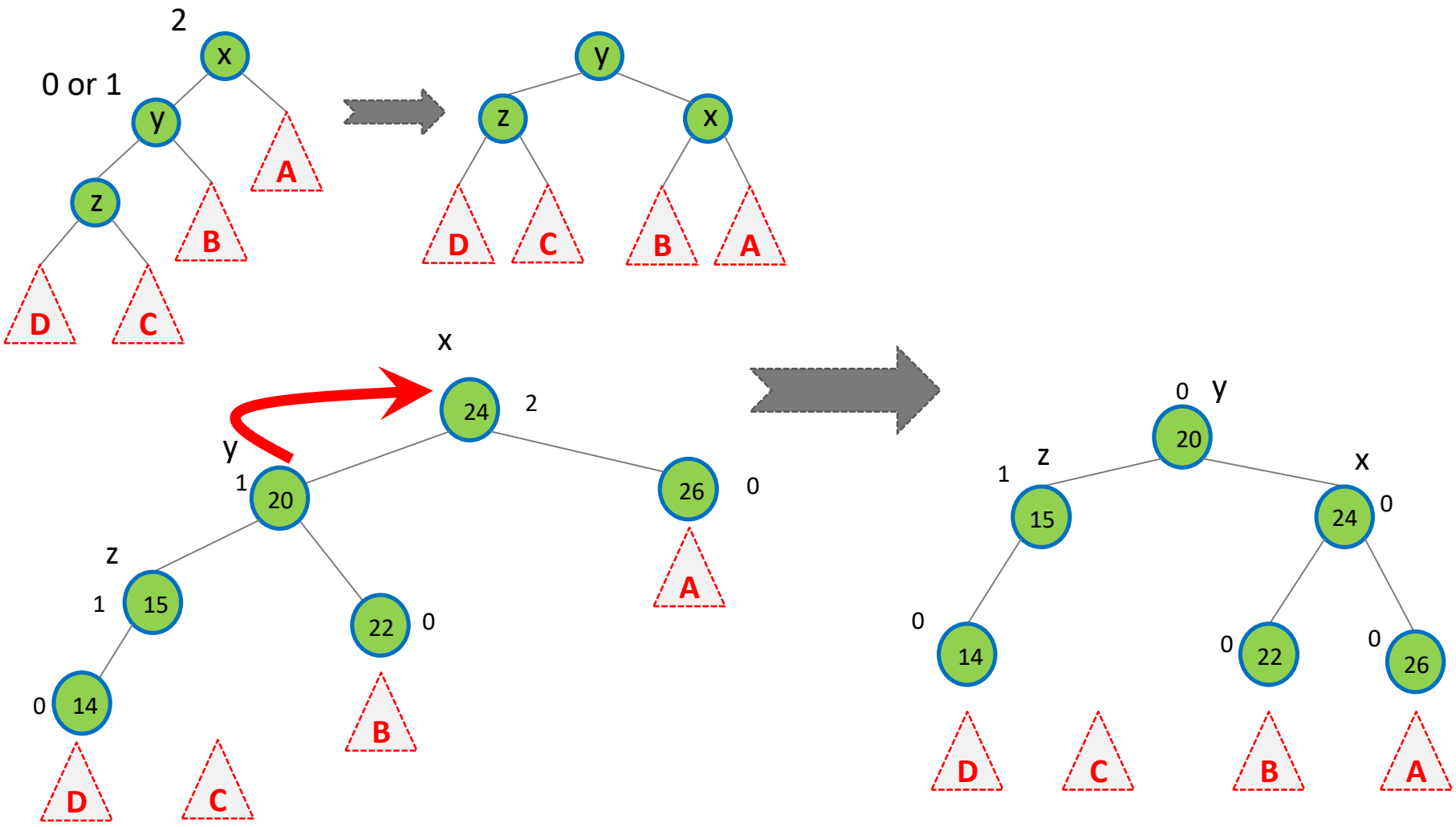




Example: Left-left case

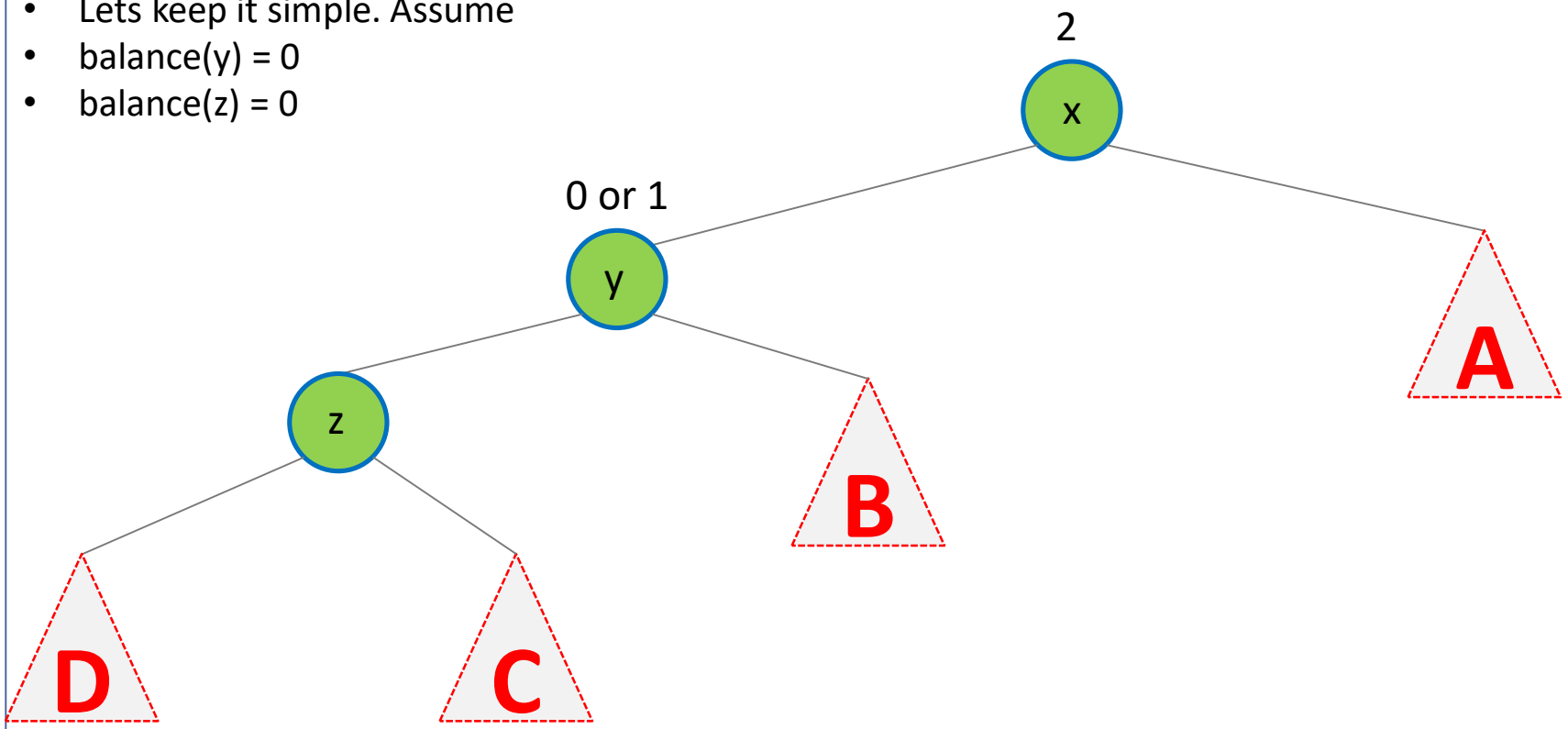


Example: Left-left case



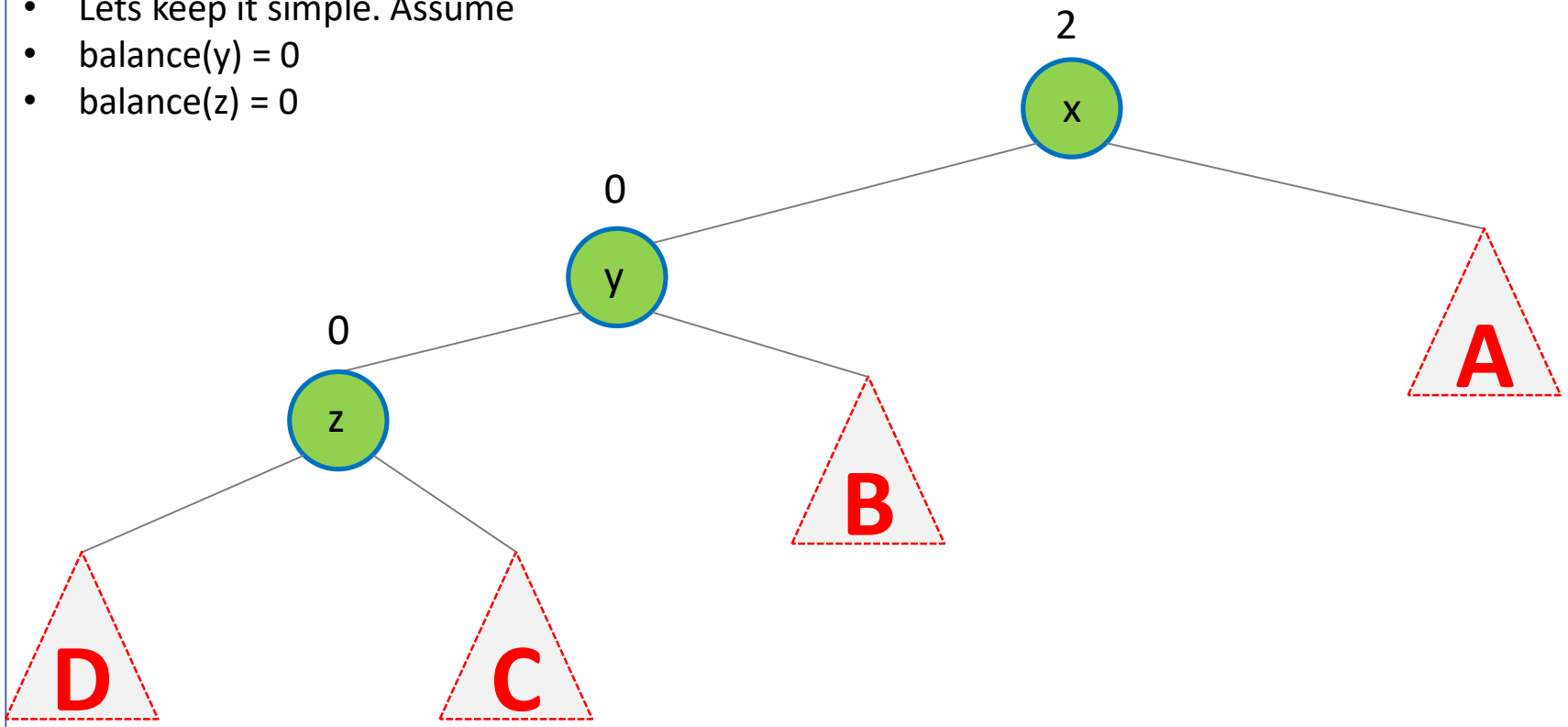
Left-Left case: Does it work?

- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0



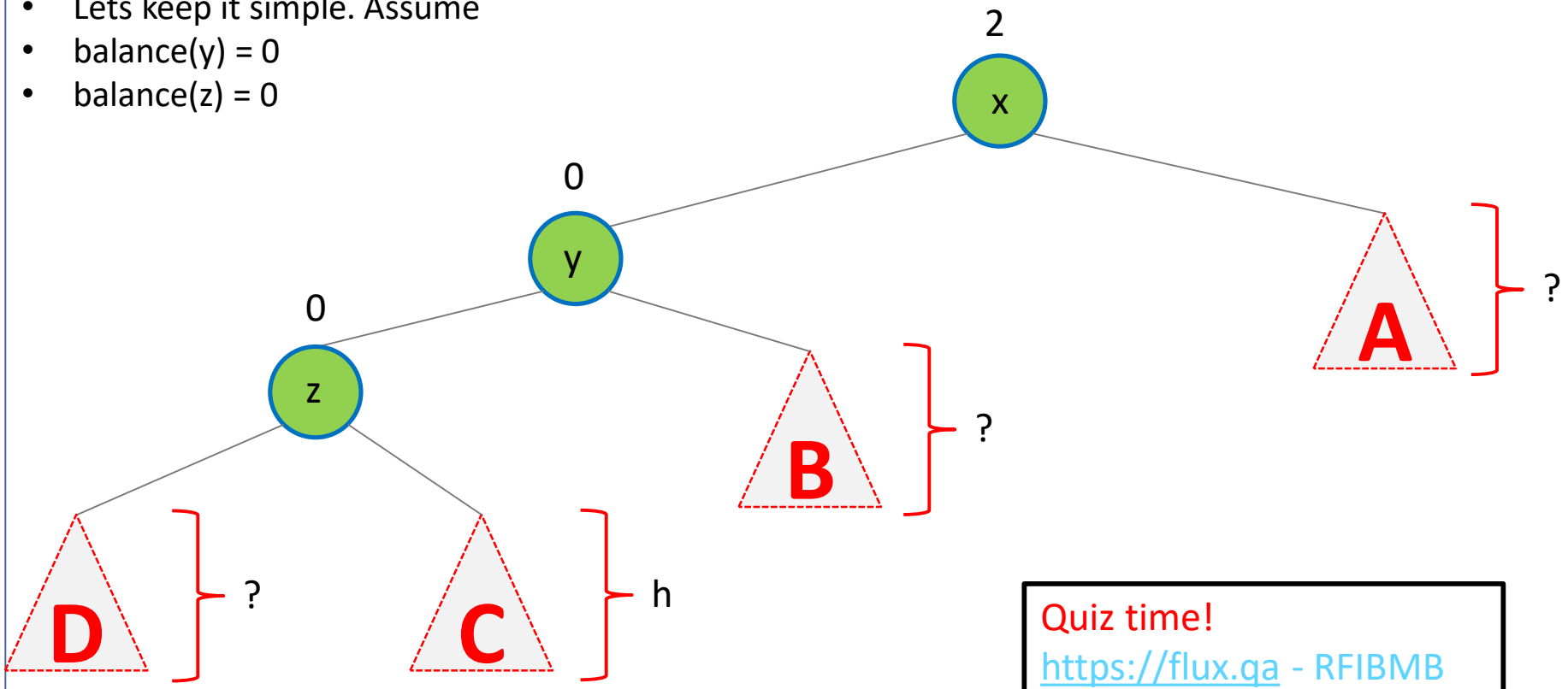
Left-Left case: Does it work?

- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0



Left-Left case: Does it work?

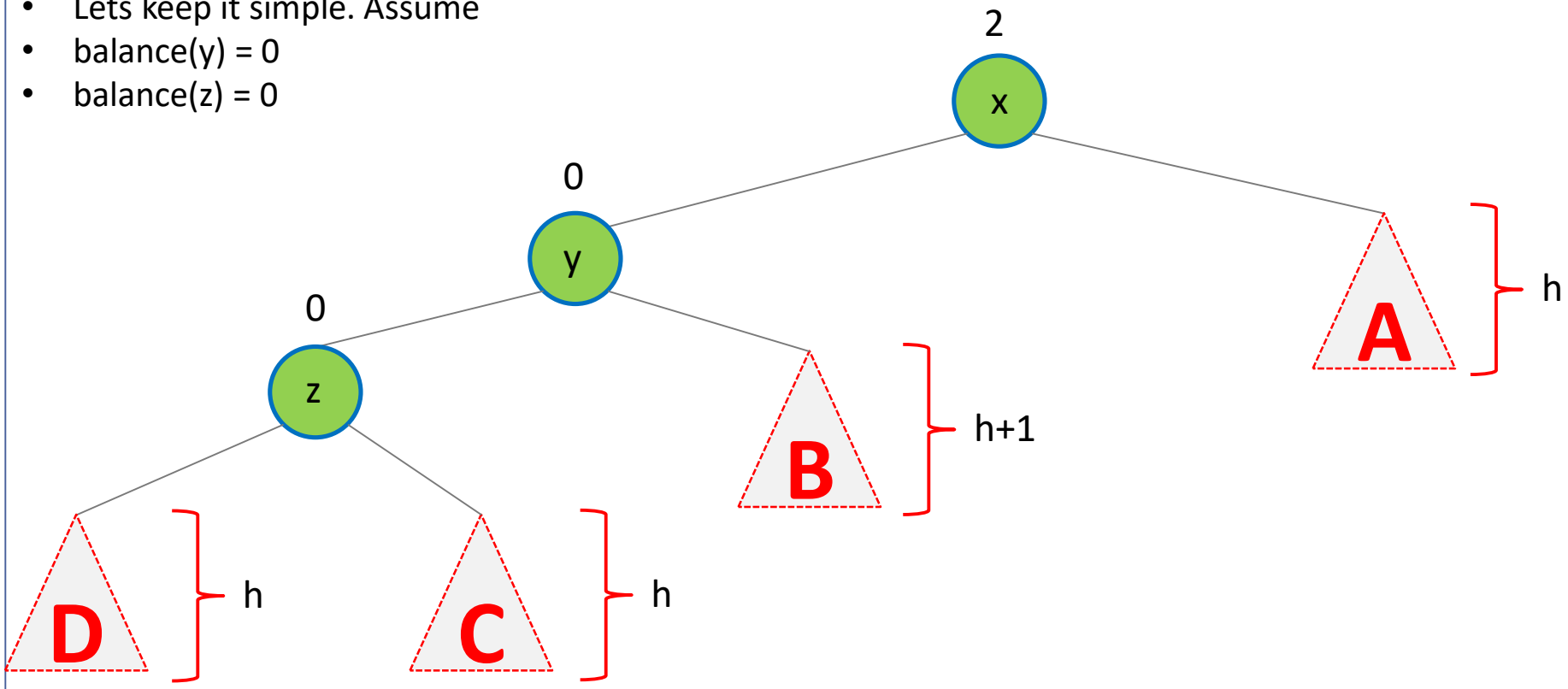
- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0



Quiz time!
https://flux.qa - RFIBMB

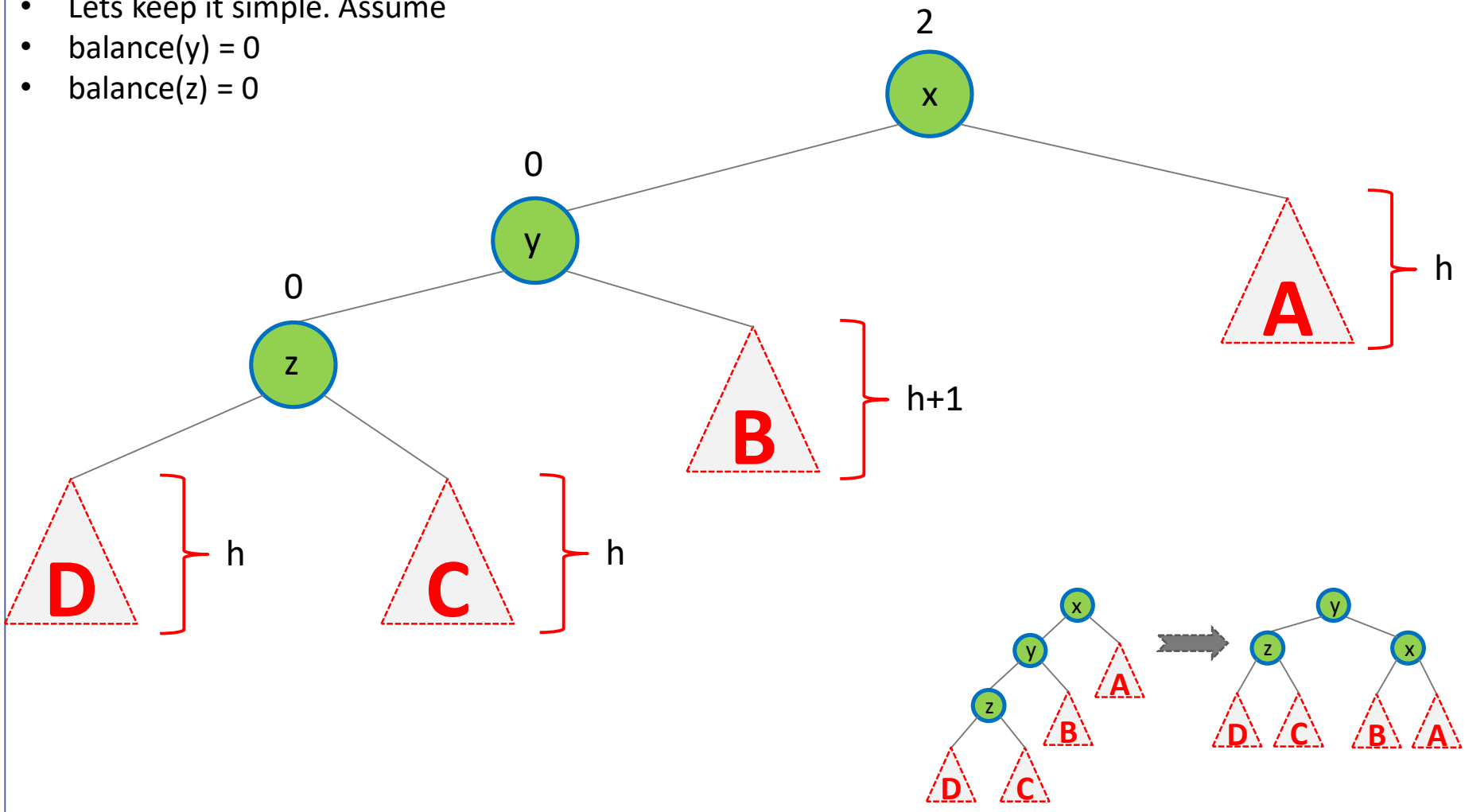
Left-Left case: Does it work?

- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0



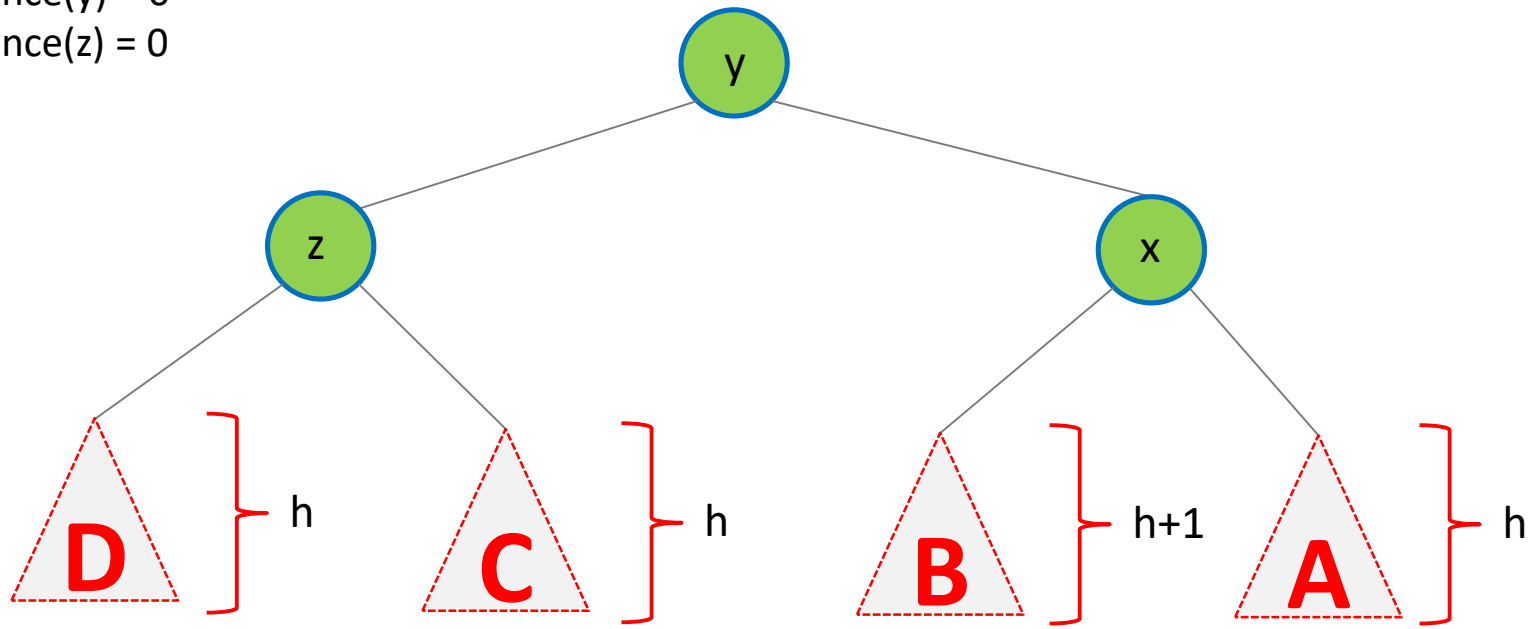
Left-Left case: Does it work?

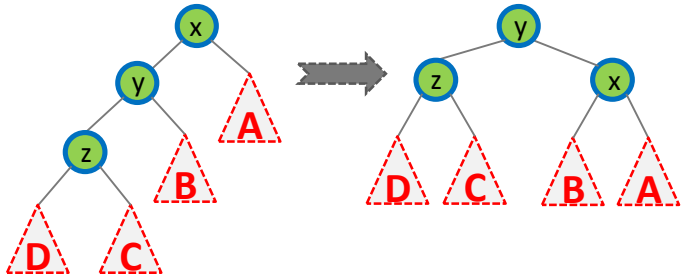
- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0



Left-Left case: Does it work?

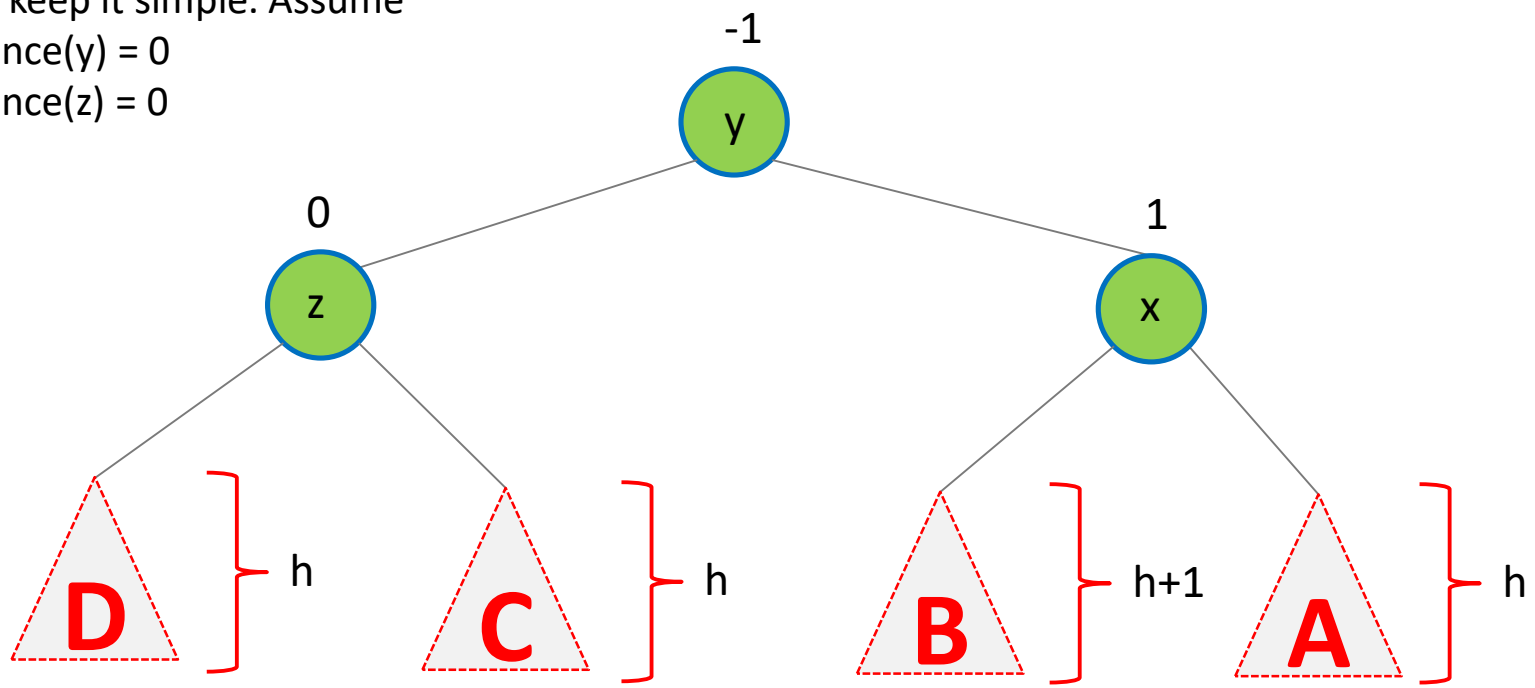
- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0

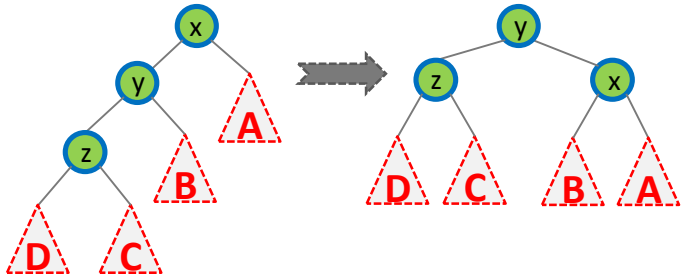




Left-Left case: Does it work?

- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0





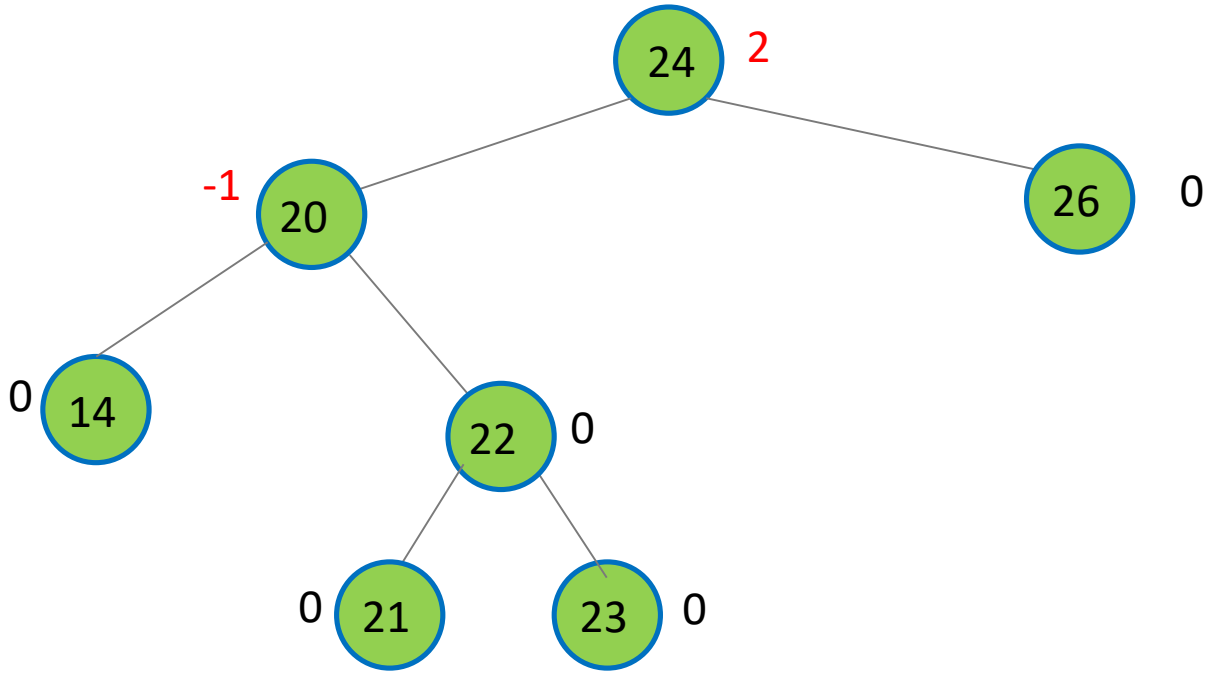
Left-Left case: Does it work?

- Now, y could have had balance factor 1
- z could have been 0, 1 or -1
- We can make similar arguments in each case, and show that the rotation gives us the balance factors we need
- Can you come up with an argument that does not require 6 cases?

Left-Right Case

A left-right case occurs when

- A node has a balance factor +2; and
- Its left child has a negative balance factor



Left-Right Case

IMPORTANT

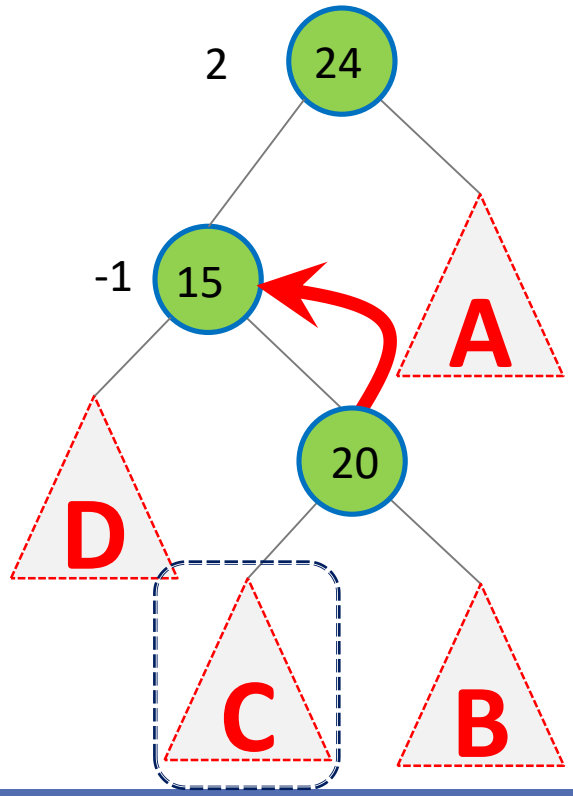
- Left-right cases are handled in two steps
- DO NOT stop in between

- After step 1, you may have created other problems with balance factors, but keep going and do step 2 before checking the state of the AVL tree

Handling Left-right case

Left Right Case

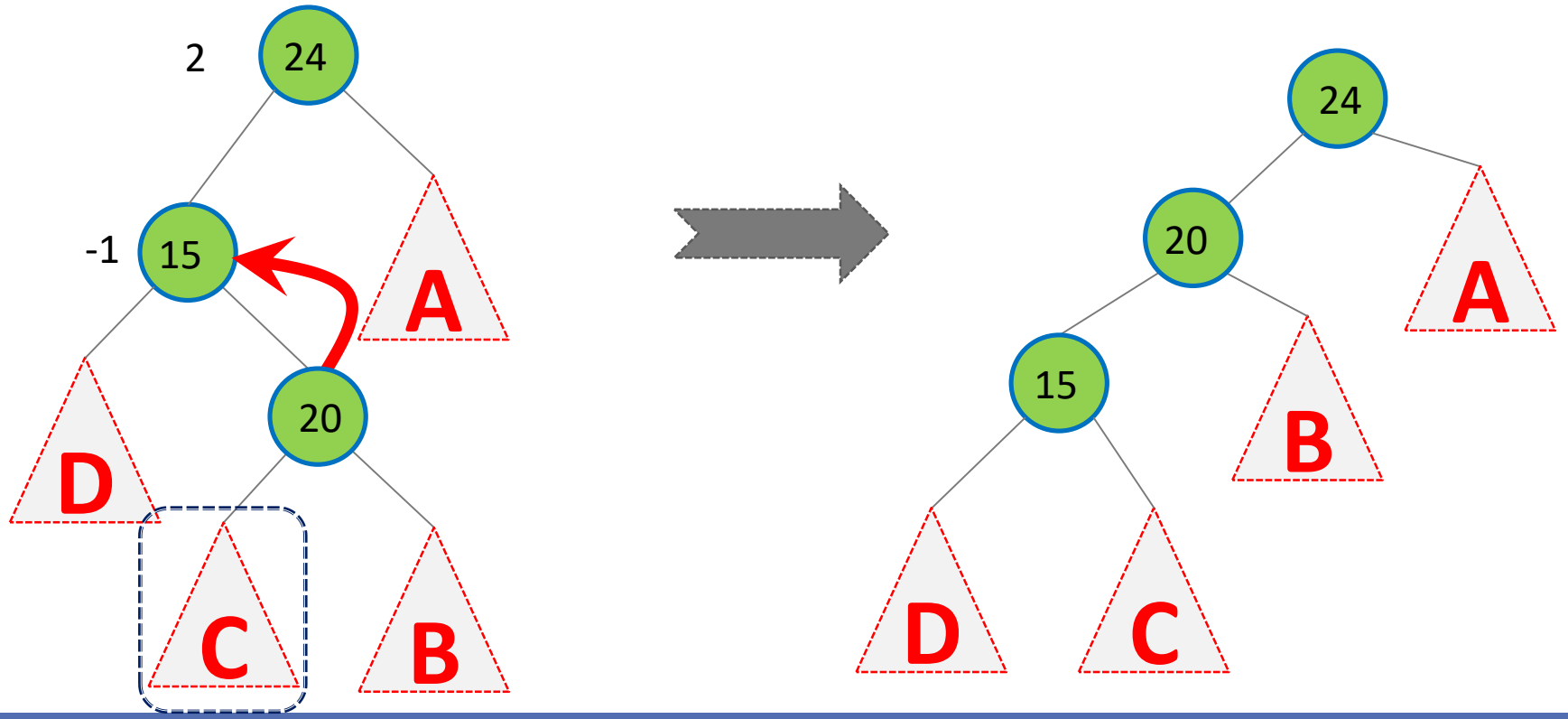
1. Convert Left Right case to Left Left case by rotating 20.



Handling Left-right case

Left Right Case

1. Convert Left Right case to Left Left case by rotating 20.

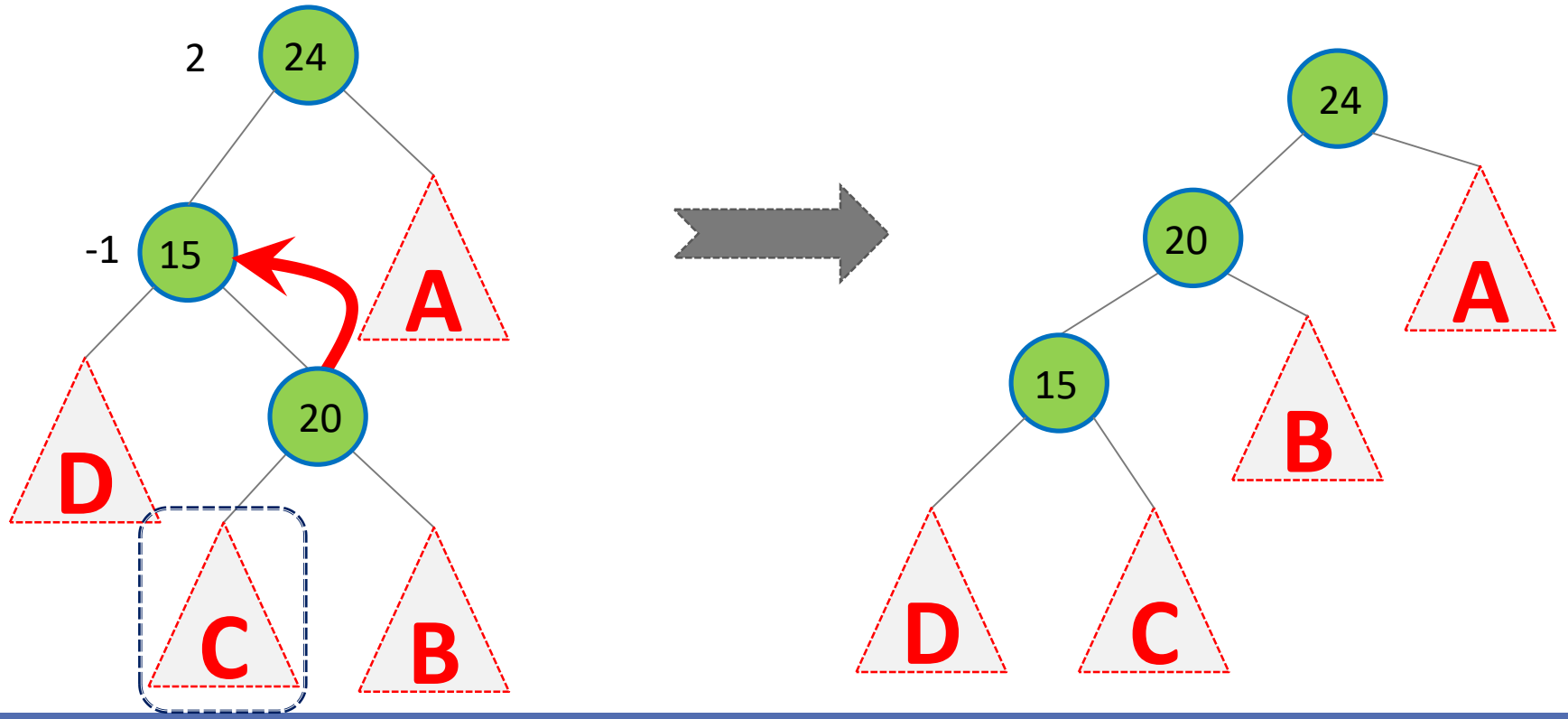


Handling Left-right case

Left Right Case

1. Convert Left Right case to Left Left case by rotating 20.

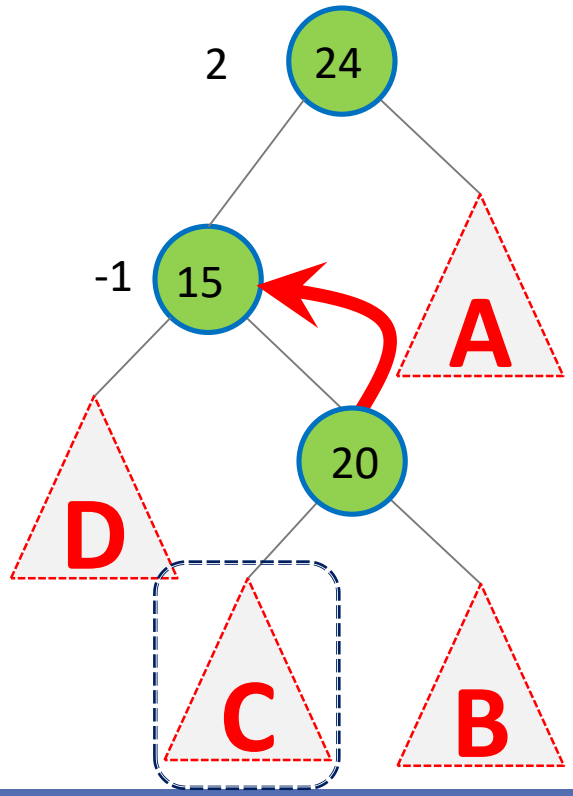
Now we have a left-left case!



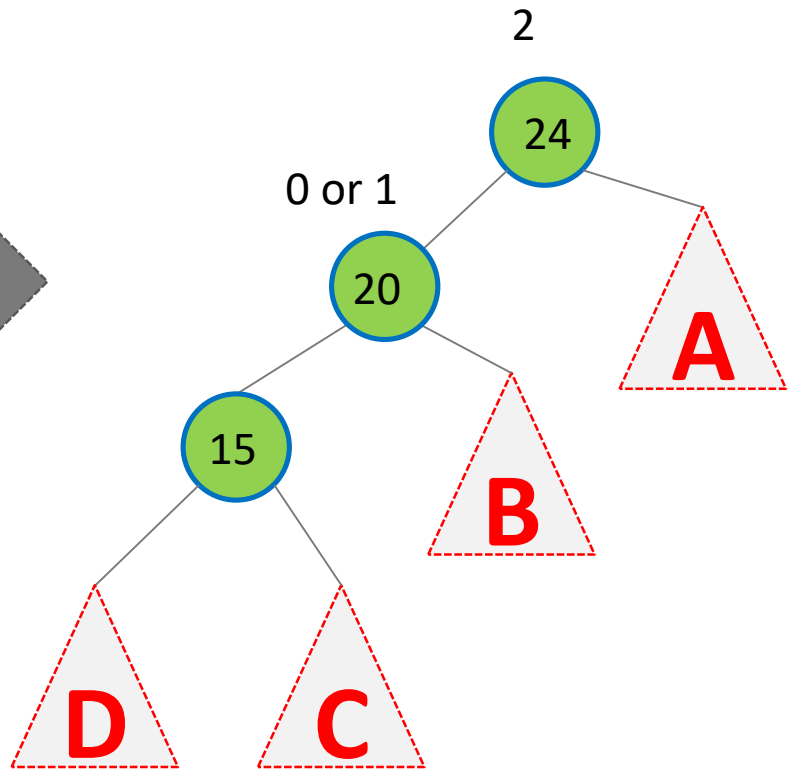
Handling Left-right case

Left Right Case

1. Convert Left Right case to Left Left case by rotating 20.
2. Handle Left Left case as described earlier



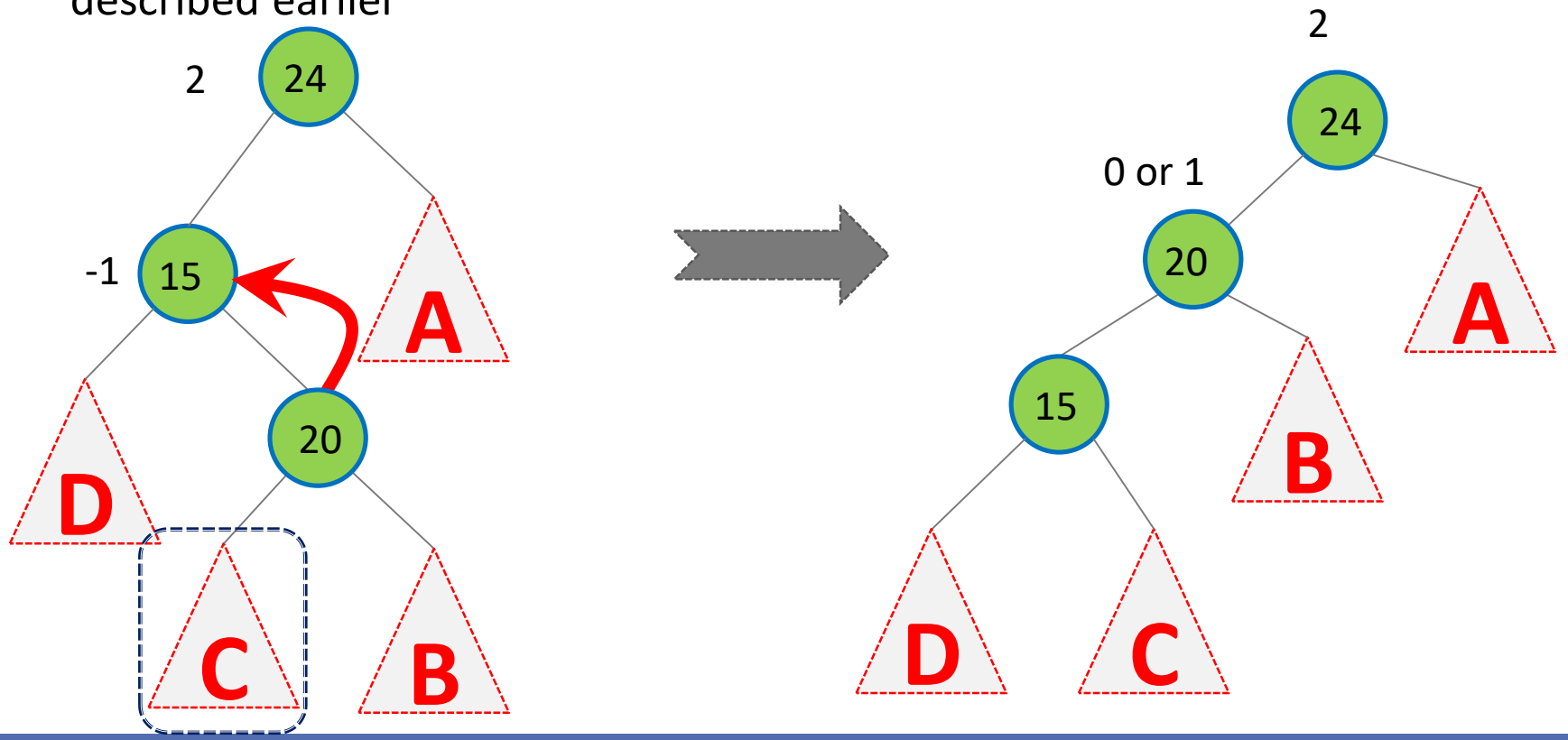
Now we have a left-left case!



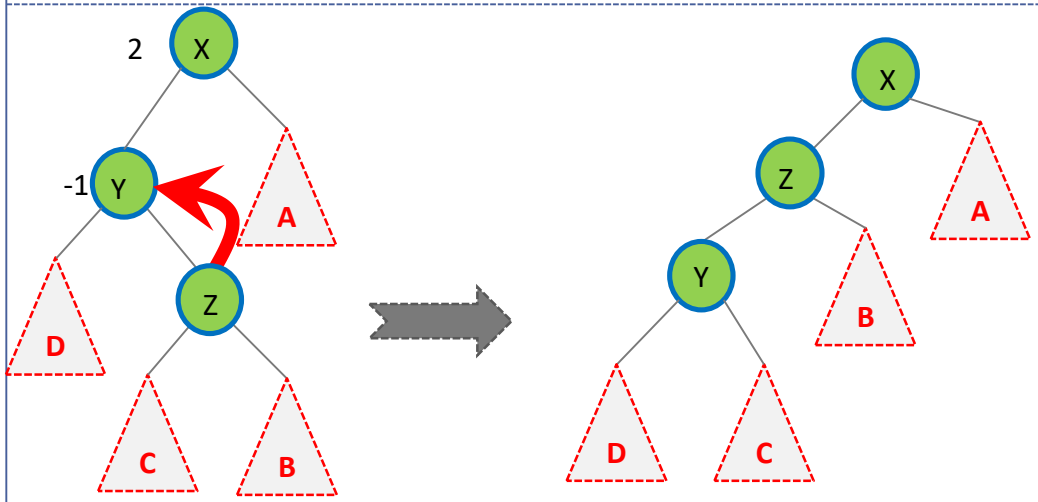
Handling Left-right case

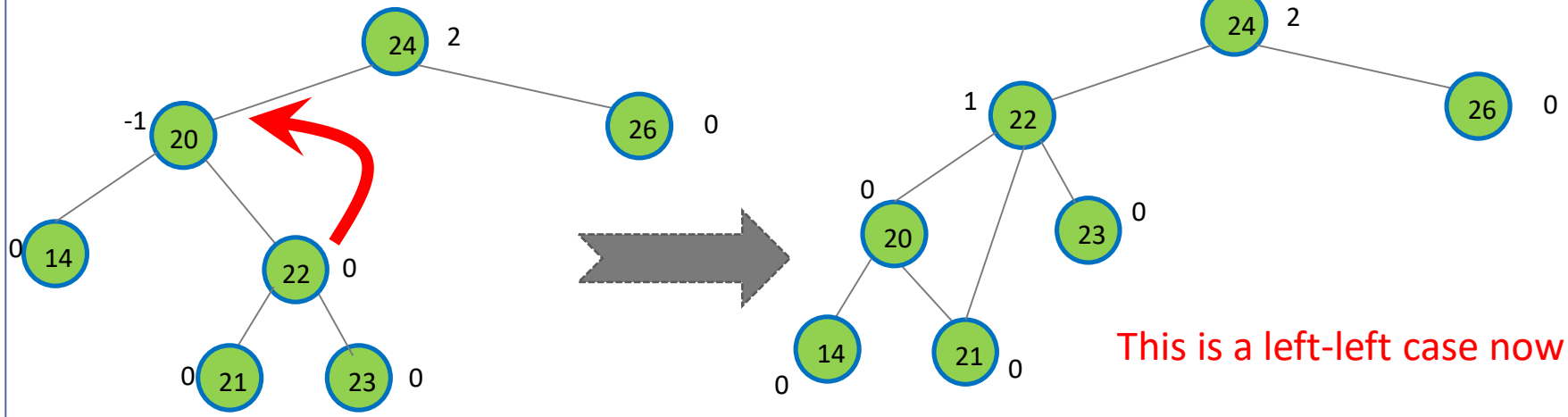
Left Right Case

1. Convert Left Right case to Left Left case by rotating 20.
2. Handle Left Left case as described earlier



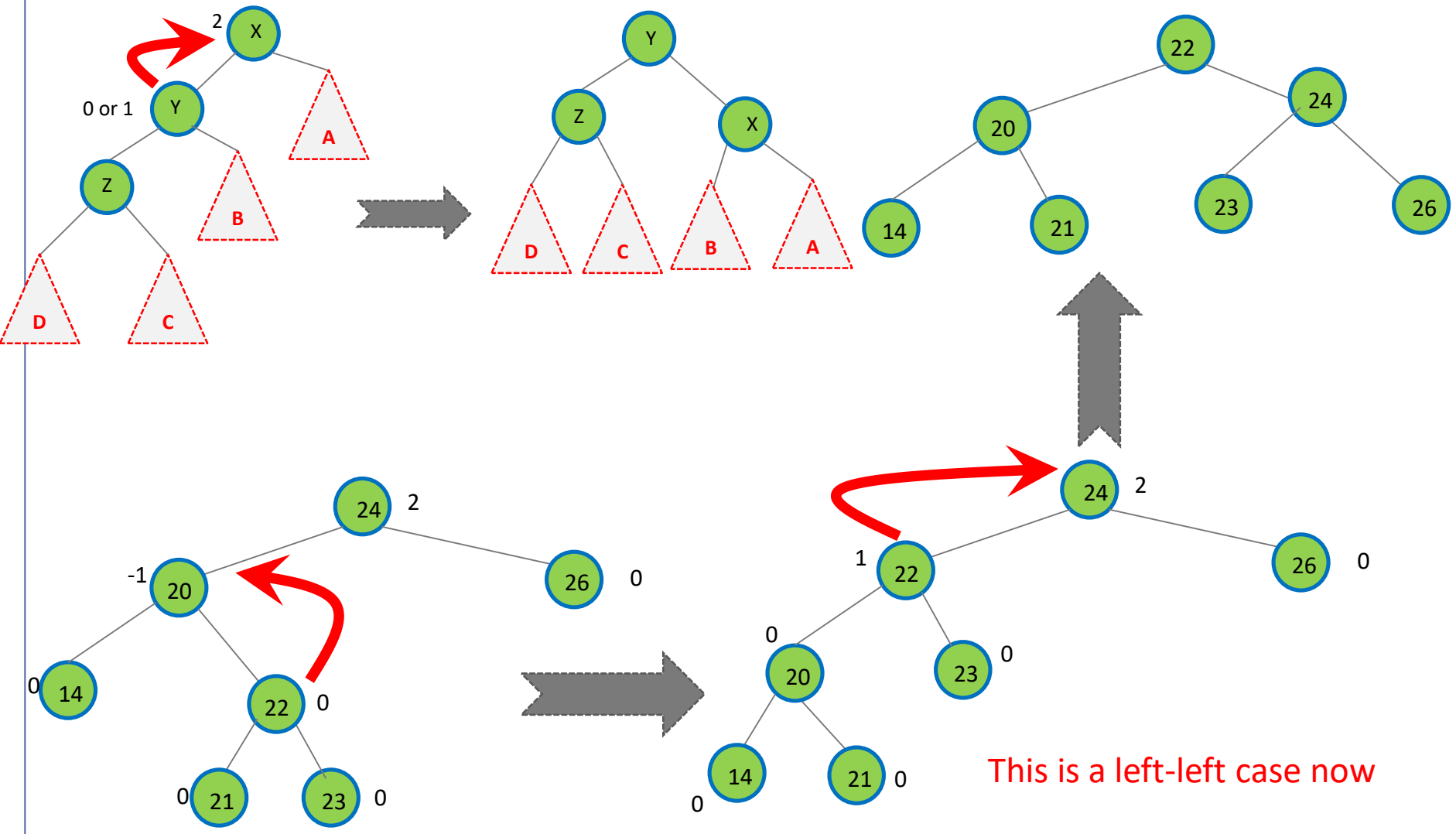
Example: Left-right case – Step 1





This is a left-left case now

Example: Left-right case – Step 2



Right-Right and Right-Left

- These two cases are mirror images of the Left-Left and Left-Right cases
- Left as an exercise

Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
   A. Introduction
   B. Balancing AVL tree
   C. Complexity Analysis
4. Hash tables

Search, Insertion, Deletion in AVL Tree

Search algorithm in AVL Tree is exactly the same as in BST

- Worst-Case time complexity
  - O(log N) because the tree is balanced (A proof of this is examined in the supplementary questions of tute 6)

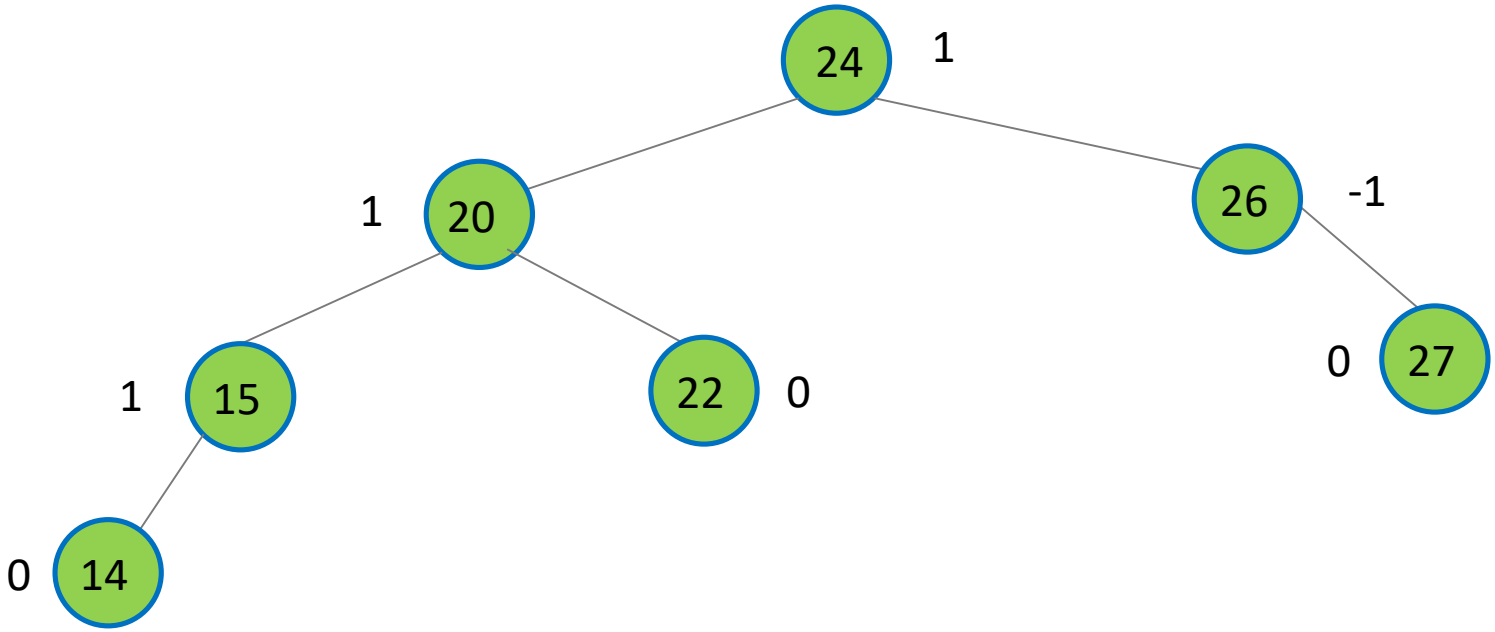
Insertion/Deletion in AVL Tree

- Insert/Delete the element in the same way as in BST (as described earlier)
- Balance the tree if it has become unbalanced (as described earlier)

Worst-case insertion/deletion time complexity: ??

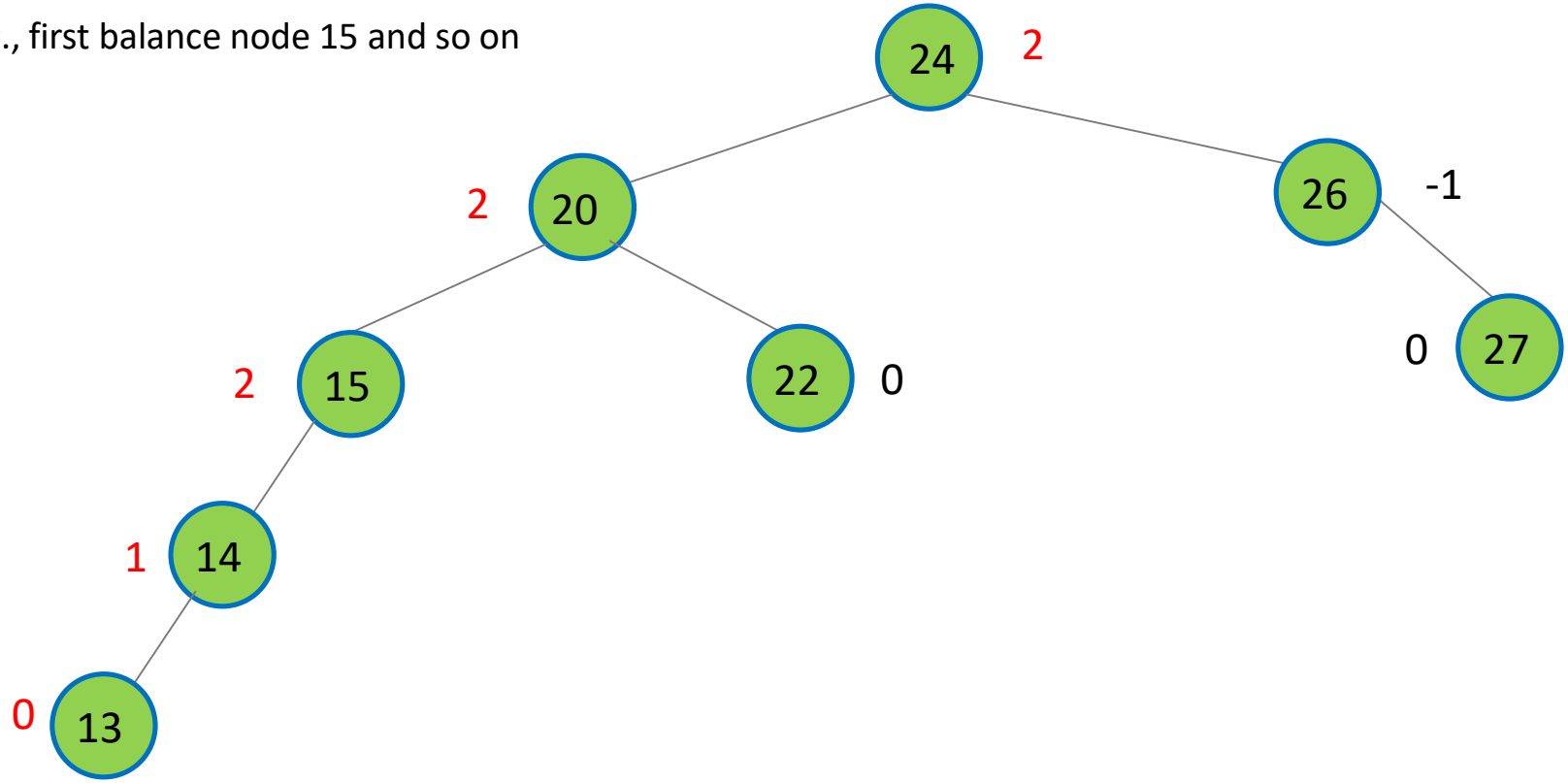
Complexity of Balancing AVL tree after insertion/deletion

- Tree is balanced before insertion/deletion
- An insertion/deletion can affect balance factor of at most O(log N) nodes
  - E.g., Insert 13



Complexity of Balancing AVL tree after insertion/deletion

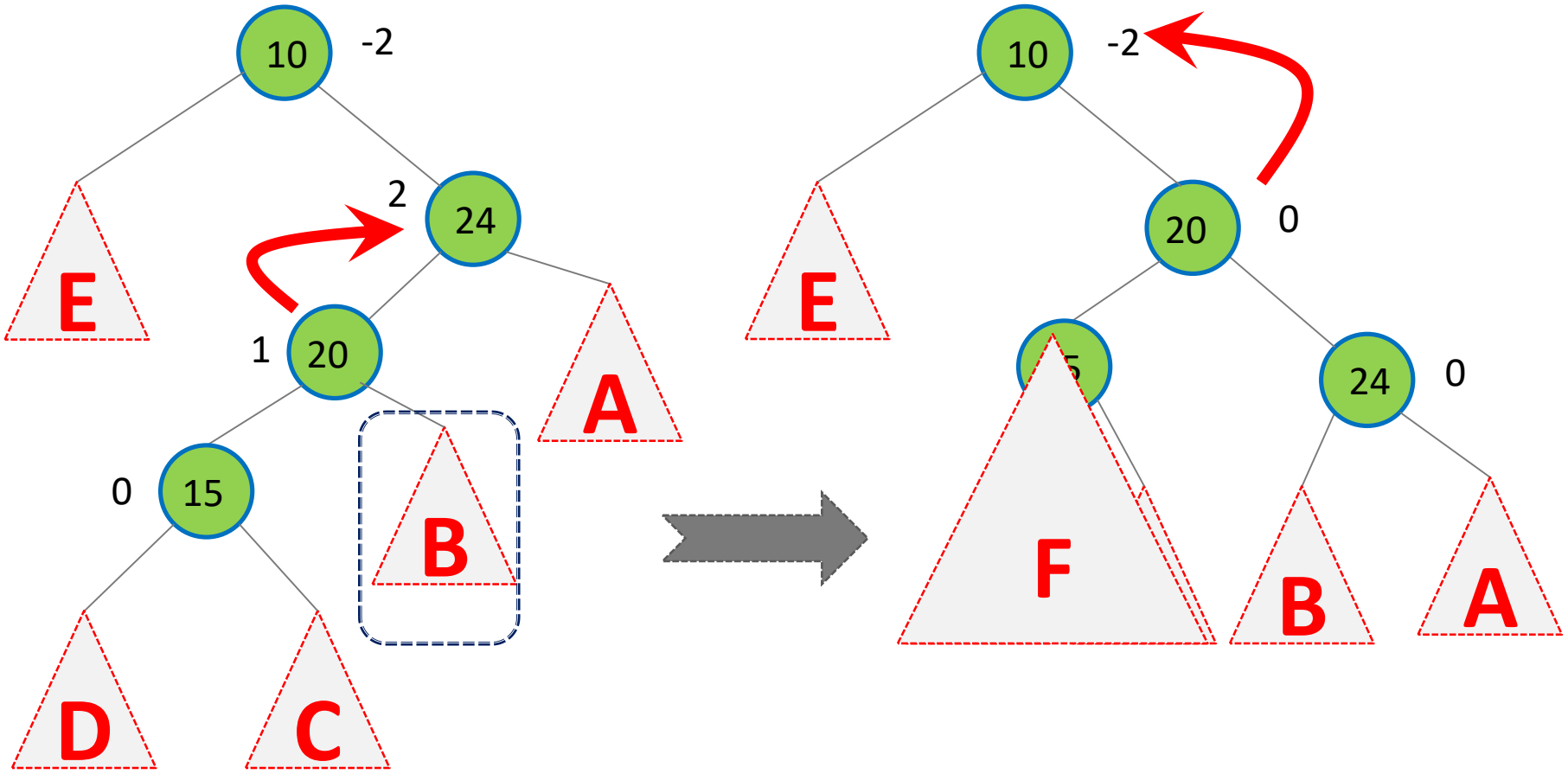
- Tree is balanced before insertion/deletion
- An insertion/deletion can affect balance factor of at most O(log N) nodes
  - E.g., Insert 13
- The balancing is done from bottom to top
  - E.g., first balance node 15 and so on



Complexity of Balancing the AVL Tree

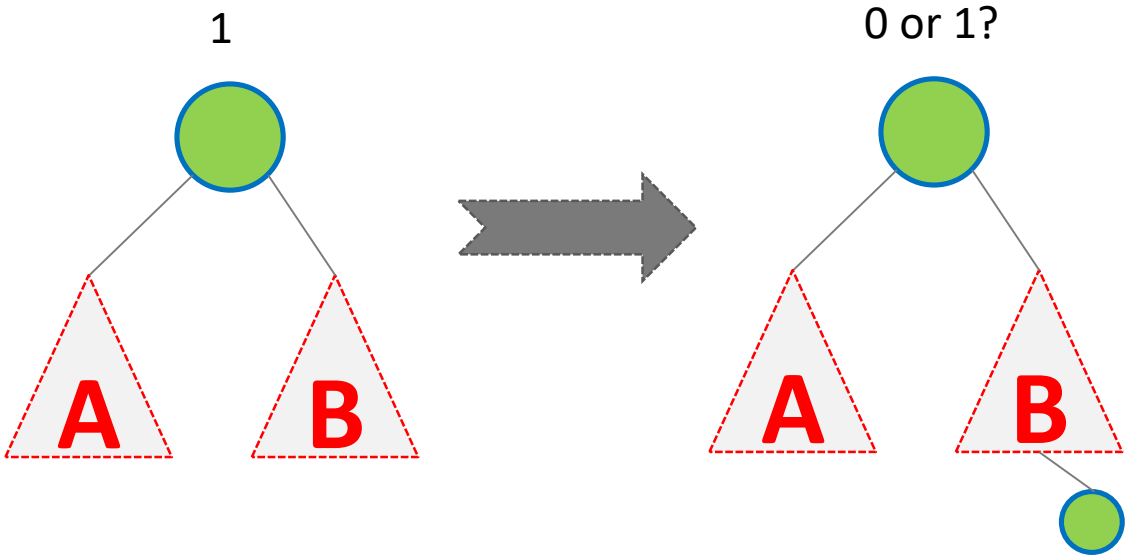
The tree is balanced in a bottom up fashion starting from the lowest node which has a balance factor NOT in {0, 1, -1}

- Balancing at each node takes constant time (1 or 2 rotations)
- We need to balance at most O(log N) nodes in the worst-case
- So total cost of balancing after insertion/deletion is O(log N) in worst-case



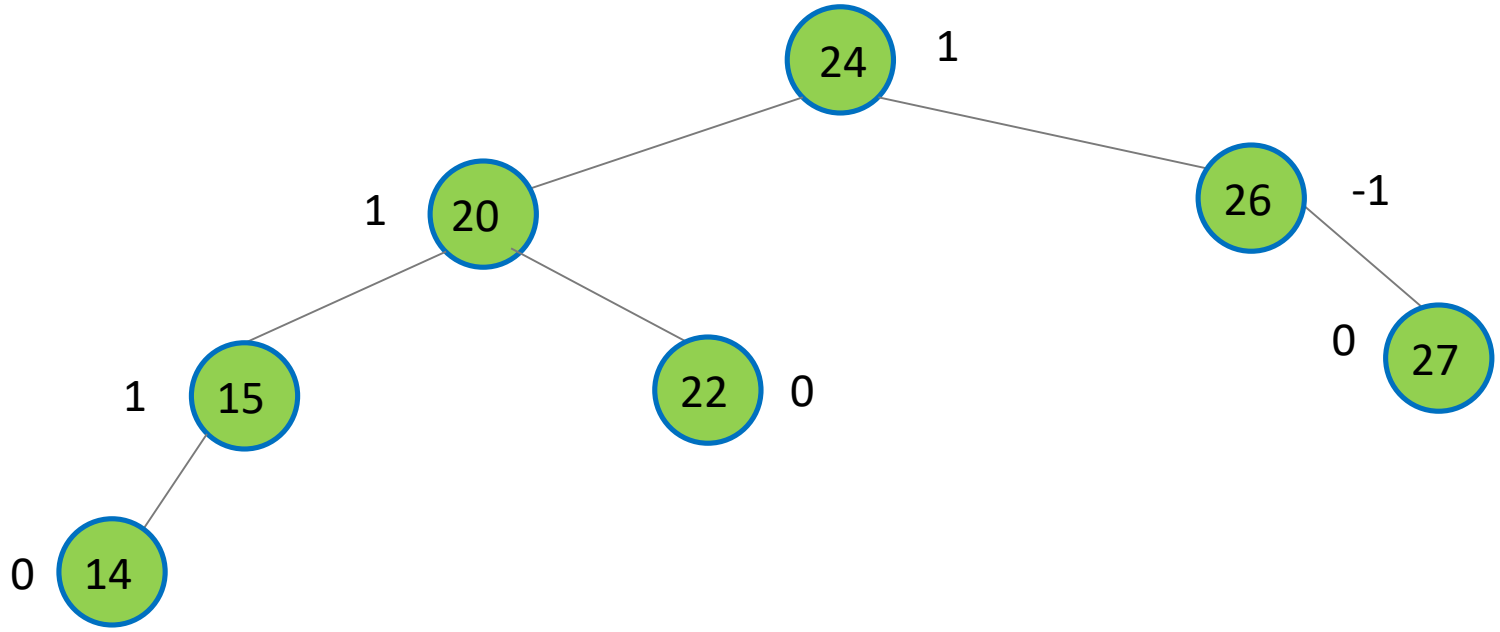
Computing balance factors

- How do we update balance factors quickly?



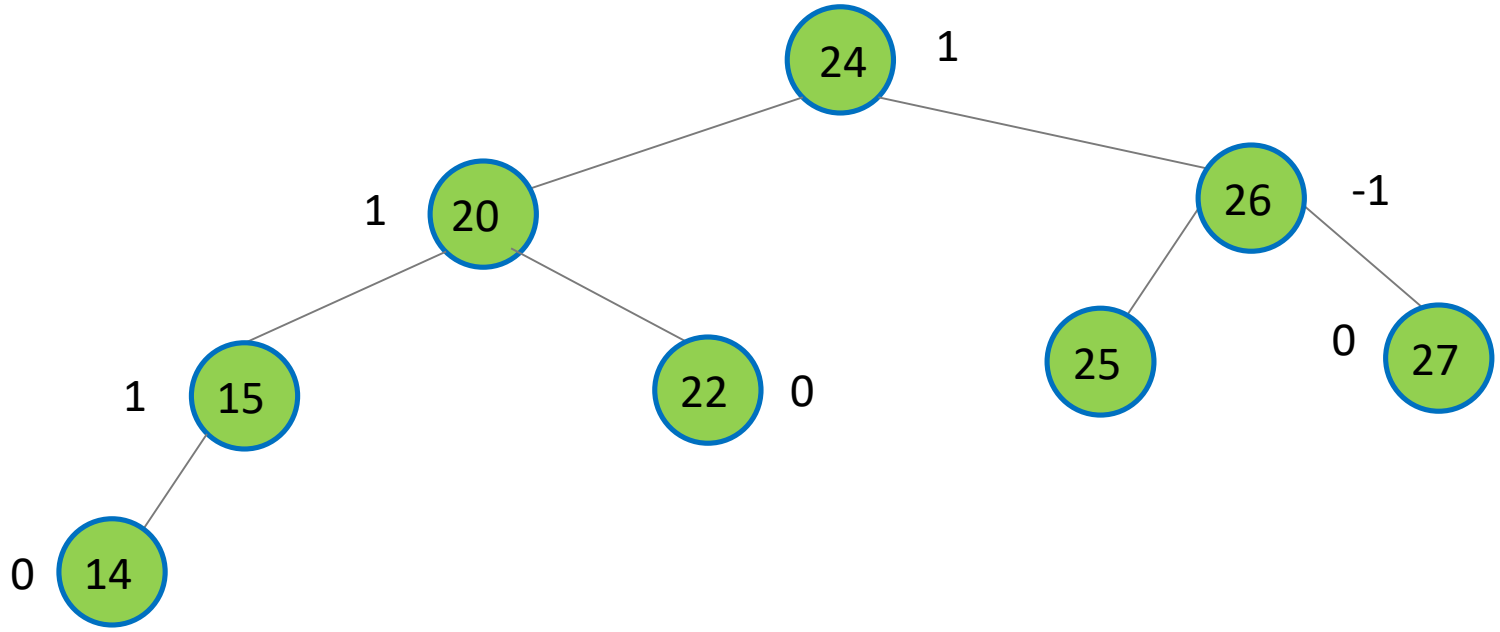
Computing balance factors

- How do we update balance factors quickly?



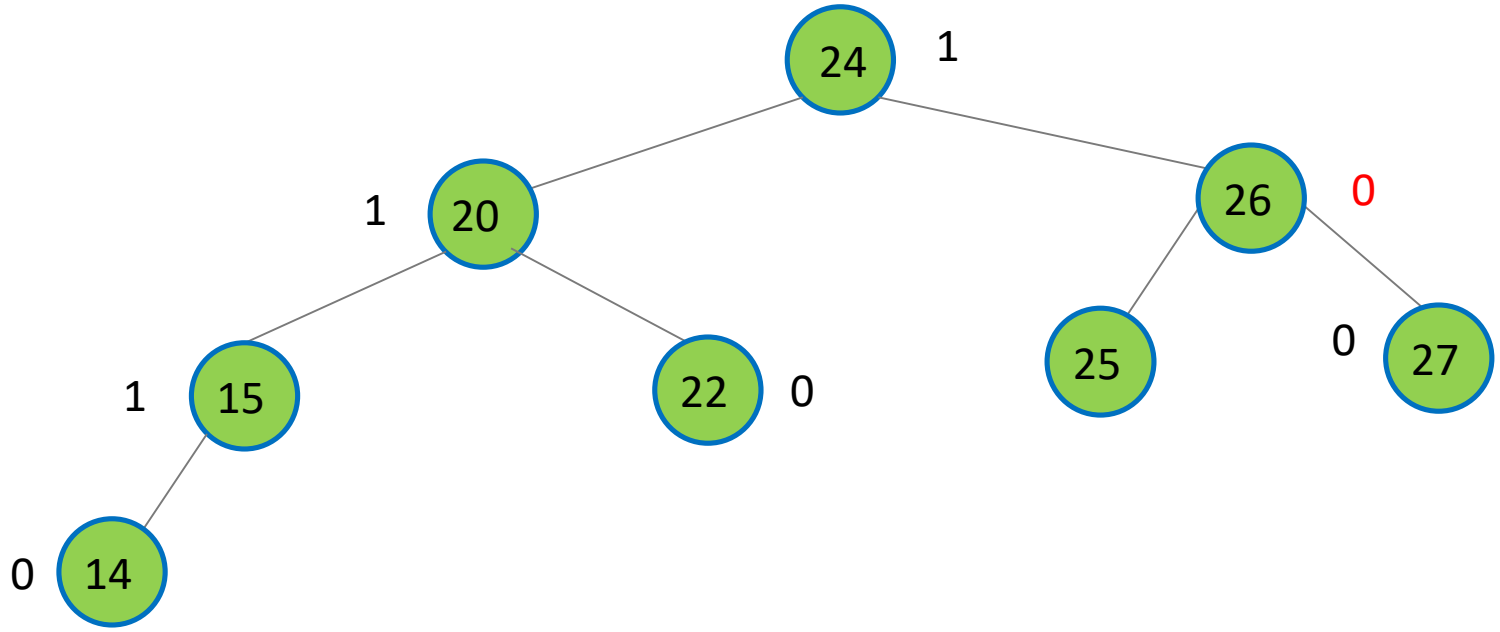
Computing balance factors

- How do we update balance factors quickly?



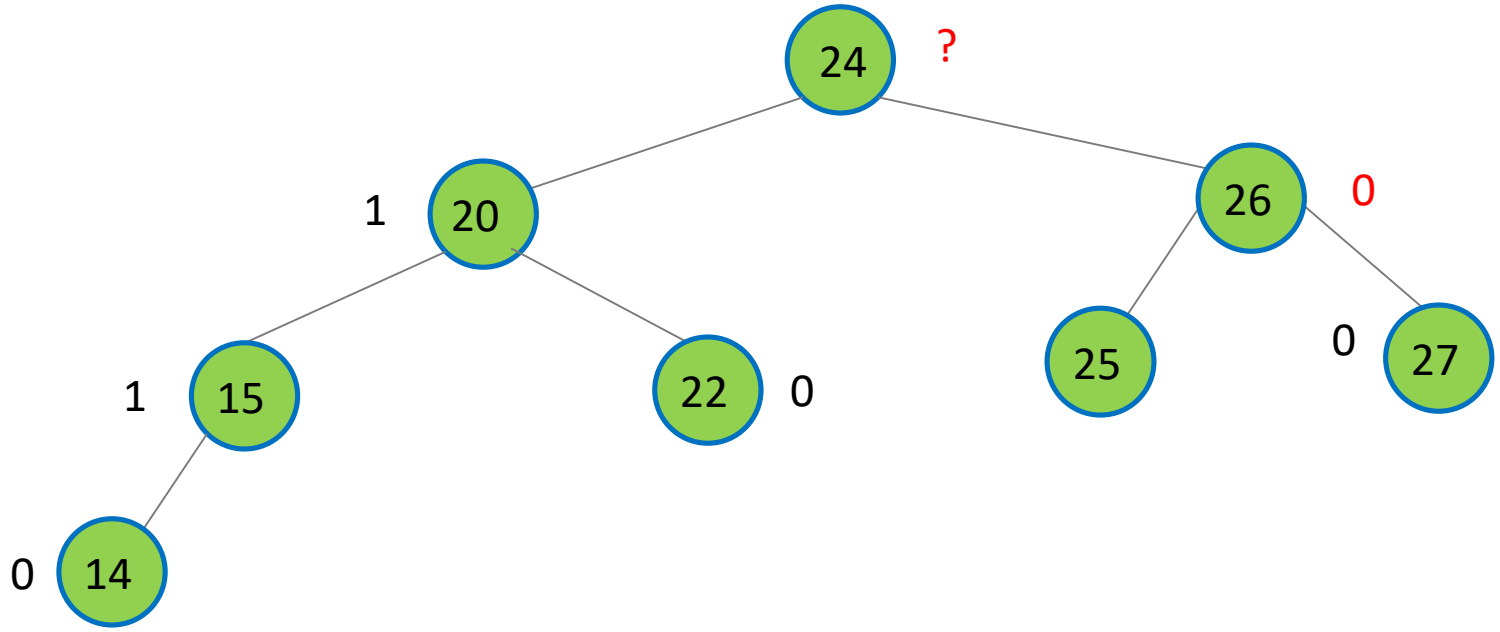
Computing balance factors

- How do we update balance factors quickly?



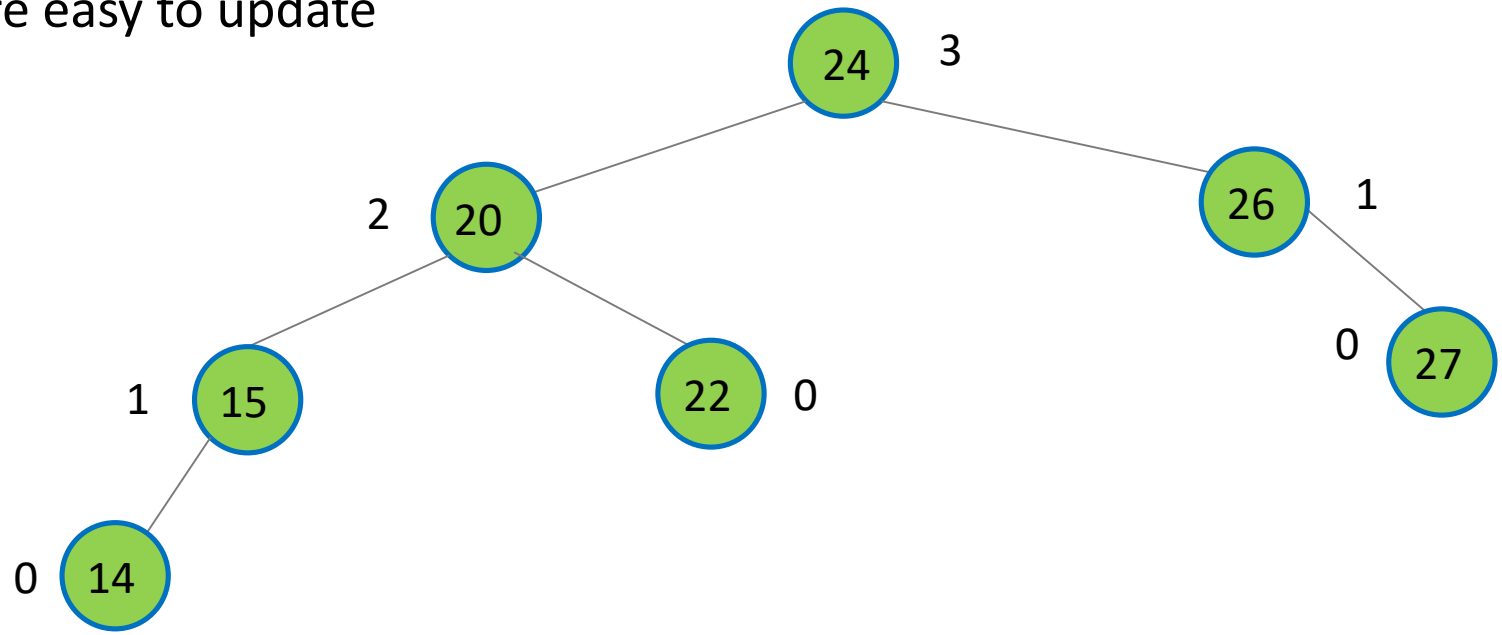
Computing balance factors

- How do we update balance factors quickly?



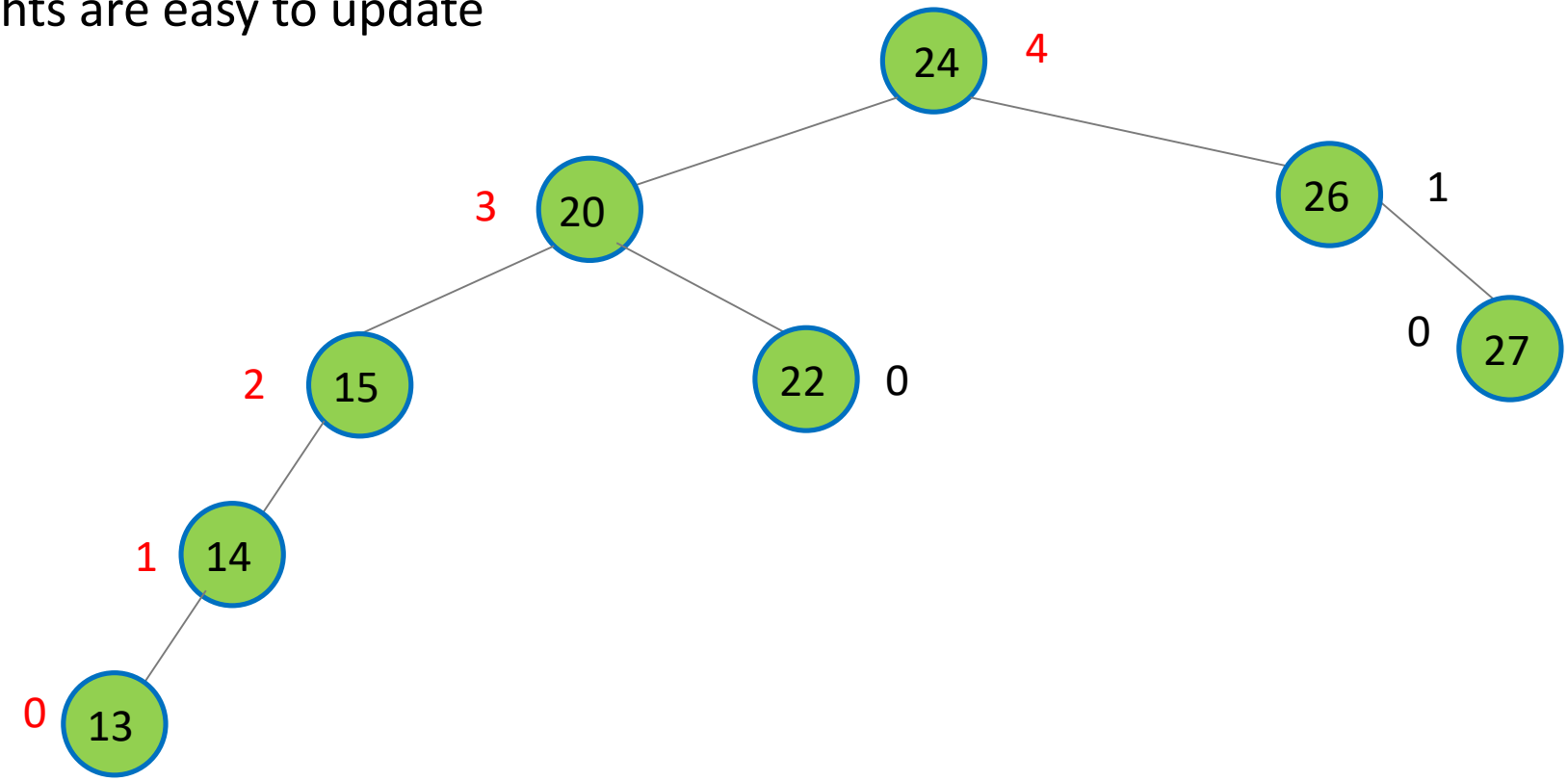
Computing balance factors

- Instead, store heights at each node
- Balance can then be computed as needed by taking height(current.left) – height(current.right)
- Heights are easy to update



Computing balance factors

- Instead, store heights at each node
- Balance can then be computed as needed by taking height(current.left) – height(current.right)
- Heights are easy to update



Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables
   A. Introduction
   B. Chaining
   C. Probing
      I. Linear Probing
      II. Quadratic Probing
      III. Double Hashing
      IV. Cuckoo Hashing

Note: Non-integer keys

- How do we index them?

- In hash tables, we **always** consider keys to be integers
- If you have a data set with non-integer keys, first apply some conversion process to the keys **(prehash)**

- This prehash might be different for different data
- Example 1111-222-333 (phone number) -> 1111222333

Direct-Addressing

Assume that we have **N** students in a class and the student IDs range from 0 to N-1. How can we store the data to delete/search/insert in O(1)-time?

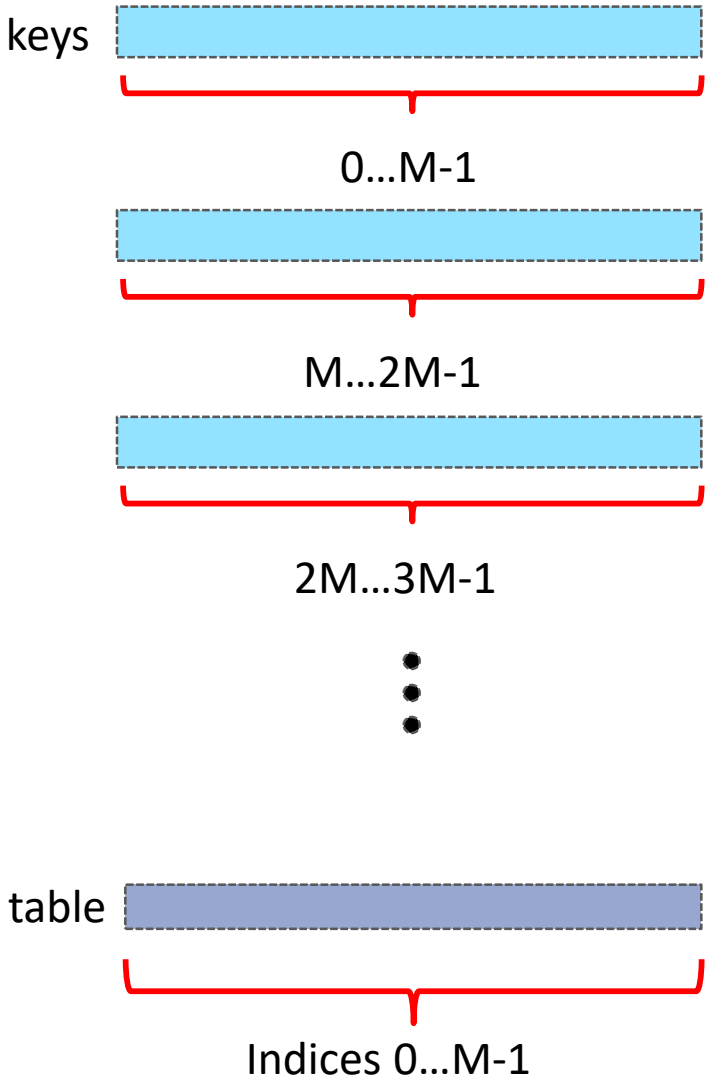
- Create an array of size N
- Store a student with ID x at index x

Note: Each student is indexed at a unique index in the array

Searching the record with ID x

- Return array[x]

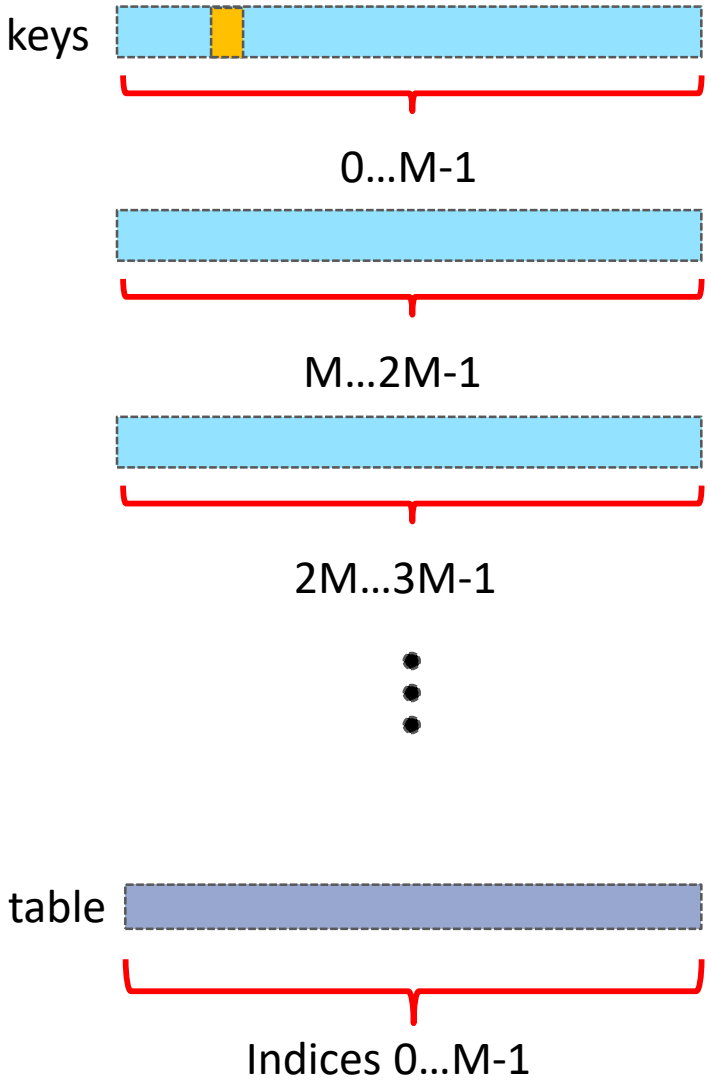
Fixing the Problem with Direct-Addressing



- **Hash function:** takes elements of a big (possibly infinite) universe and maps them to some finite range
- i.e. indices in [0..M-1]

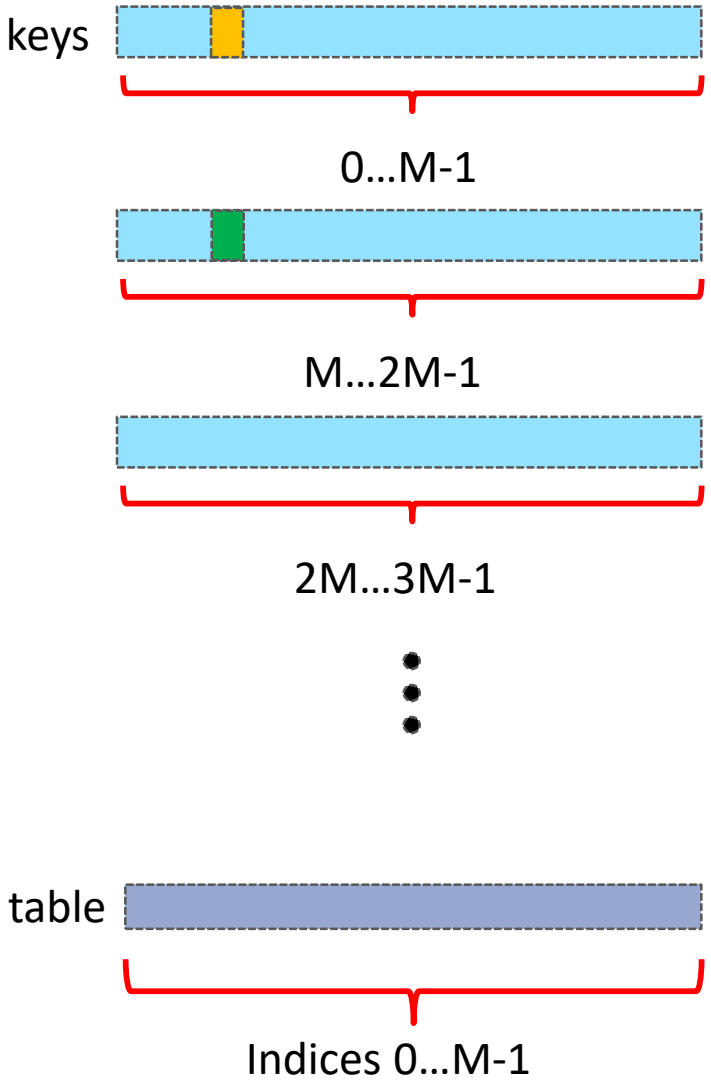
- One way to do this in practice is to use modulus (%)
- **Note that just taking keys % m is a bad hash function!**
- **a%m** = the remainder when **a** is divided by **m**
- Always gives a value in the range [0..m-1]

Collisions



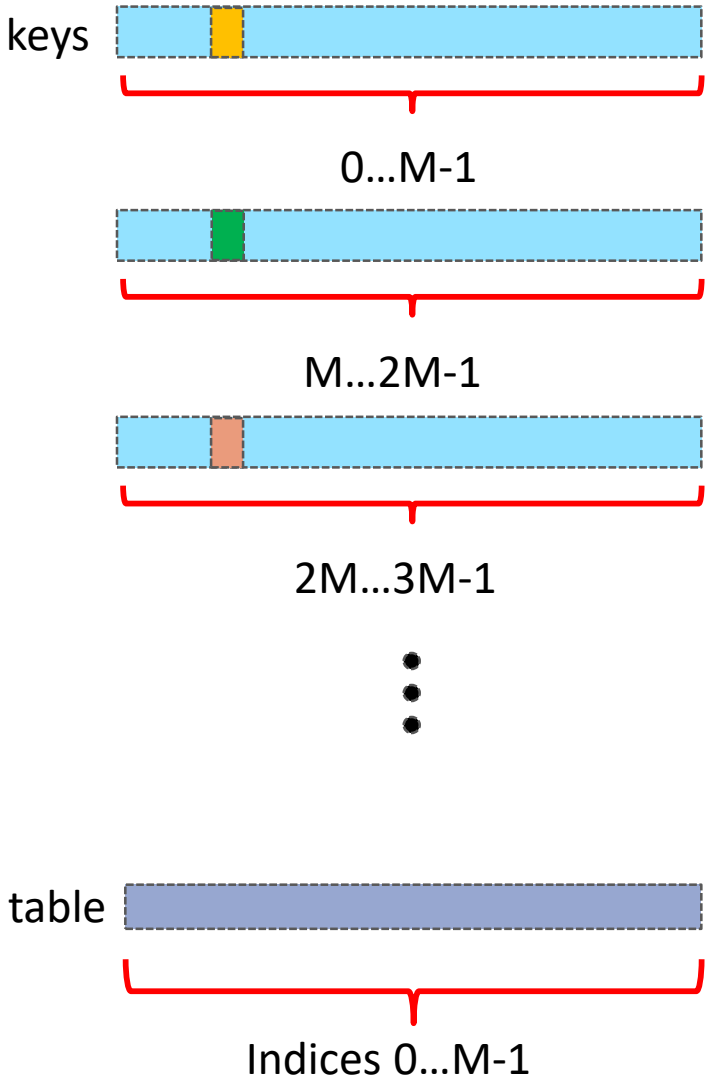
- A consequence of having more keys than indices in our table

Collisions



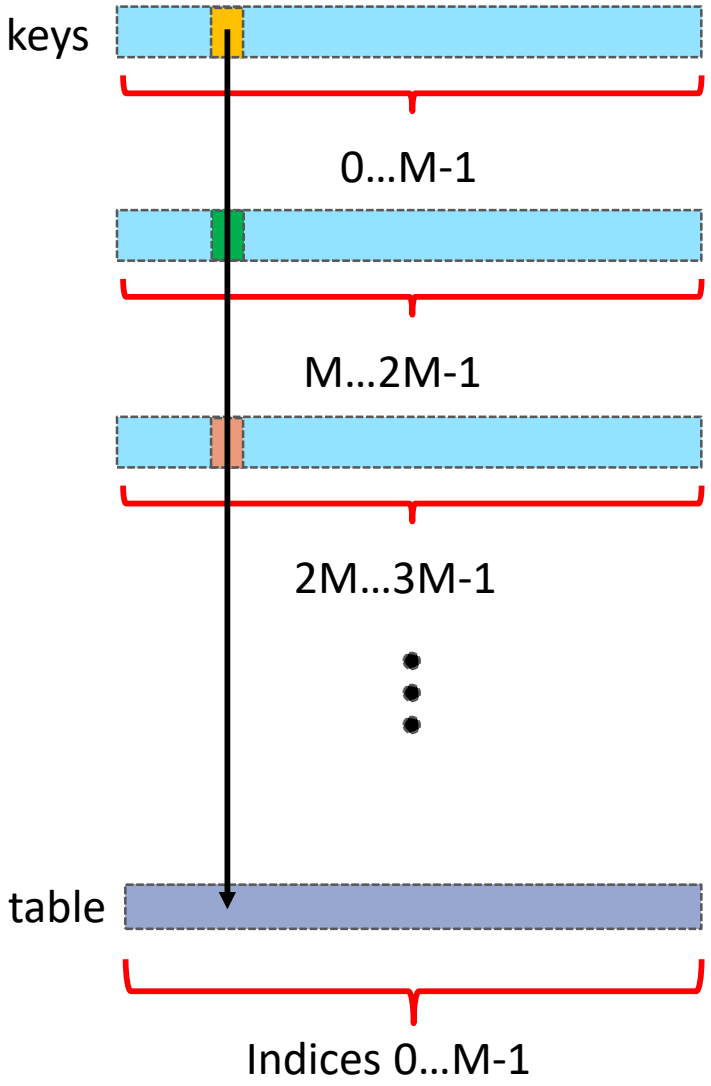
- A consequence of having more keys than indices in our table

Collisions



- A consequence of having more keys than indices in our table

Collisions



- A consequence of having more keys than indices in our table
- Many values “belong” (are hashed to) in the same slot

Collision probability

- N items
- Table size = M
- Chance of collision?

- Similar to a famous “paradox”, the birthday paradox

- **How many people do you need in a room before 2 of them share a birthday?**

- In this situation, “table size” is 365, and N is the number of people

Collision probability

- **How many people do you need in a room before 2 of them share a birthday?**

Quiz time!
https://flux.qa - RFIBMB

- In this situation, "table size" is 365, and N is the number of people

$$Prob(no\ collision) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \cdots \times \frac{(366 - N)}{365}$$

$$Prob(no\ collision) = \frac{365!}{365^N (365 - N)!}$$

Visit https://pudding.cool/2018/04/birthday-paradox/ for an interactive explanation of the birthday paradox

Probability of Collision

- prob(collision) = 1 – prob(no collision)
- The probability of collision for 23 people is ~ 50%
- The probability of collision for 50 people is 97%
- The probability of collision for 70 people is 99.9%

Handling Collisions

- Two strategies to address collisions!
- **Chaining:** for each hash value, we have a separate data structure which stores all items with that hash value
- **Probing:** items which collide get inserted somewhere else in the array
- **Note:** These two general strategies have many other names, but they are confusing

Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables
   A. Introduction
   B. Chaining
   C. Probing
      I. Linear Probing
      II. Quadratic Probing
      III. Double Hashing
      IV. Cuckoo Hashing

Chaining

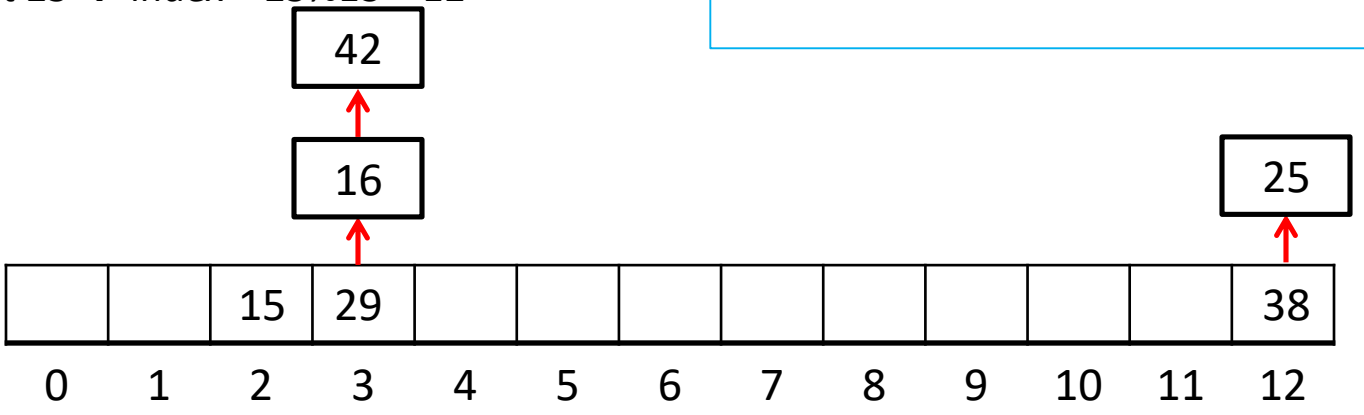
- If there are already some elements at hash index
  - Add the new element in a list at array[index]
- Example: Suppose the hash table size M is 13 and hash function is key % 13.
  - Insert 29 → index = 29%13 = 3
  - Insert 38 → index = 38%13 = 12
  - Insert 16 → index = 16%13 = 3
  - Insert 15 → index = 15%13 = 2
  - Insert 42 → index = 42%13 = 3
  - Insert 25 → index = 25%13 = 12

Lookup/searching an element:

- index = hash(key)
- Search in list at Array[index]

Deleting an element:

- Search the element
- Delete it



Chaining

- Assume that M is the size of hash table and N is the number of records already inserted.
- Assume that our hash functions distributes our keys uniformly over our hash table
- Assume that the load of the table is < c

- We know that there are at most cM items in the table
- These cM items are distributed among M “chains”
- The average chain has c < 1 items in it
- The **average** time complexity of an insert operation is O(1)

Complexity of resizing - intuition

- What about the insert that triggers a resize?

Table size	Total work for insertion (half tablesize)	Total work for resize (double the table, reinsert
m	m/2	2m + m/2
2m	m	4m + 3m/2
4m	2m	8m + 7m/2
…	…	…
$2^i m$	$2^{i-1} m$	$2^{i+1} m + O(2^i m)$

Complexity of resizing - intuition

- Imagine spreading out the resize work over the insertions
- This concept is called "amortized analysis" (not examinable)

Table size	Total work for insertion	Total work for resize
m	m/2	2m + m/2
2m	m	4m + m
4m	2m	8m + 2m
…	…	…
$2^i m$	$2^{i-1} m$	$2^{i+1} m + (2^{i-1} m)$

- The amortized cost of each insert is O(1), even though most of the work occurs on one specific insert (the one which triggers a resize)

Other data structures as “chains”

- We could use anything as our “chains”
- We want something with fast insert, lookup and delete
- I wonder what that could be?

Quiz time!
https://flux.qa - RFIBMB

- Balanced binary search trees!
- Or what about…
- Hash tables? (Look at the tute problem on FKS hashing)

Probing

- In probing schemes, each index in the hash table contains at most one element.
- How to avoid collision in this case?
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
  - Cuckoo Hashing

Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables
   A. Introduction
   B. Chaining
   C. Probing
      I. Linear Probing
      II. Quadratic Probing
      III. Double Hashing
      IV. Cuckoo Hashing

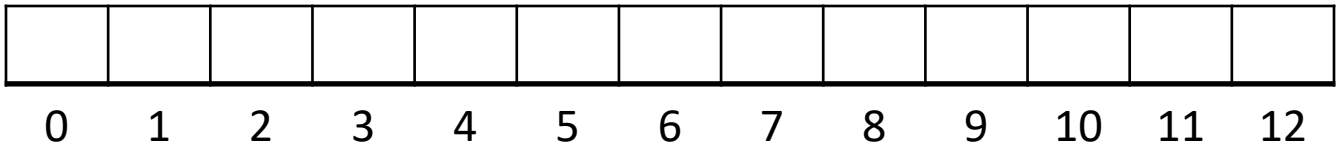
Linear Probing

- In case of collision, sequentially look at the next indices until you find an empty index

- **h(k, i) = h'(k) + i**
- h'(k) is some hash function
- **i** is how many times we have probed

- For example, suppose h'(k)= k % 12

```
// psuedocode for linear probing
index = hash(key)
i = 1
while array[index] is not empty and i != M
    index = (hash(key) + i) % M
    i ++
if i != M
   insert element at array[index]
```

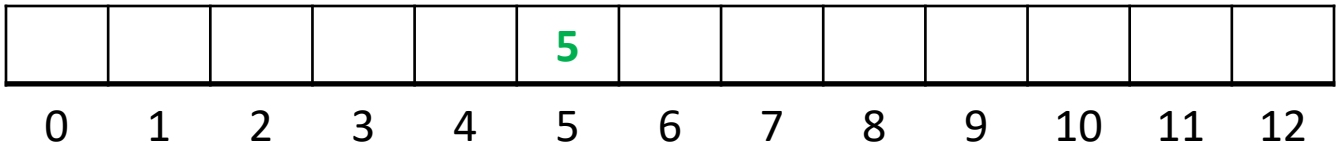


Linear Probing

- h'(k) = k % 12
- h(k,i) = k % 12 + i

- Insert 5

```
// psuedocode for linear probing
index = hash(key)
i = 1
while array[index] is not empty and i != M
    index = (hash(key) + i) % M
    i ++
if i != M
   insert element at array[index]
```

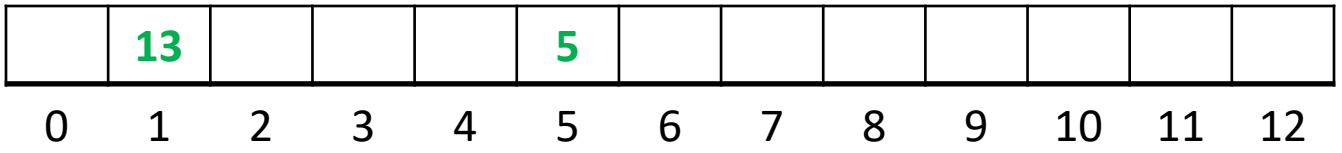


Linear Probing

- h'(k) = k % 12
- h(k,i) = k % 12 + i

- Insert 5
- Insert 13: 13%12 = 1

```
// psuedocode for linear probing
index = hash(key)
i = 1
while array[index] is not empty and i != M
    index = (hash(key) + i) % M
    i ++
if i != M
   insert element at array[index]
```

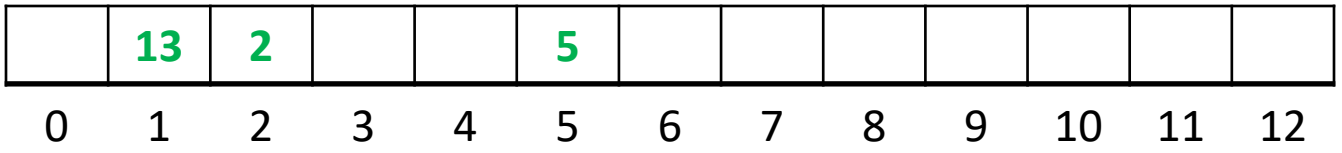


Linear Probing

- h'(k) = k % 12
- h(k,i) = k % 12 + i

- Insert 5
- Insert 13: 13%12 = 1
- Insert 2

```
// psuedocode for linear probing
index = hash(key)
i = 1
while array[index] is not empty and i != M
    index = (hash(key) + i) % M
    i ++
if i != M
   insert element at array[index]
```

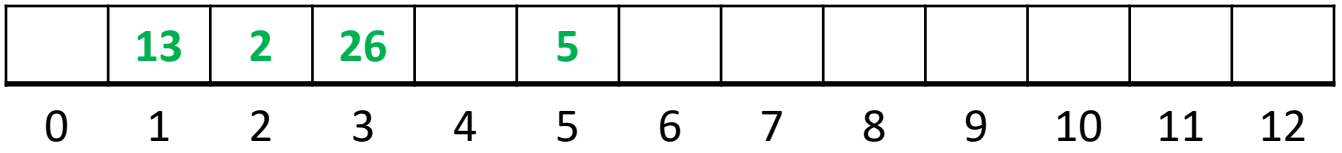


Linear Probing

- h'(k) = k % 12
- h(k,i) = k % 12 + i

- Insert 5
- Insert 13: 13%12 = 1
- Insert 2
- Insert 26: 26%12 = 2, probe once to 3

```
// psuedocode for linear probing
index = hash(key)
i = 1
while array[index] is not empty and i != M
    index = (hash(key) + i) % M
    i ++
if i != M
   insert element at array[index]
```

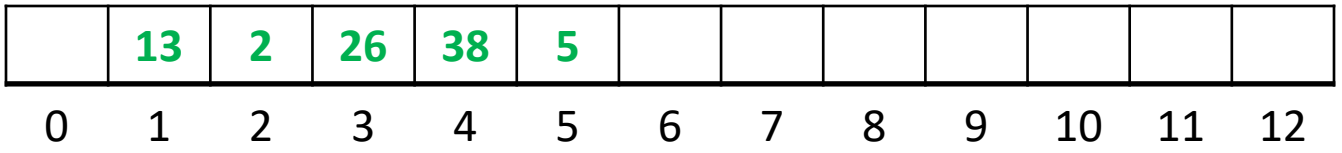


Linear Probing

- h'(k) = k % 12
- h(k,i) = k % 12 + i

- Insert 5
- Insert 13: 13%12 = 1
- Insert 2
- Insert 26: 26%12 = 2, probe once to 3
- Insert 38: 38%12 = 2, probe twice to 4

```
// psuedocode for linear probing
index = hash(key)
i = 1
while array[index] is not empty and i != M
    index = (hash(key) + i) % M
    i ++
if i != M
   insert element at array[index]
```



Linear Probing

Searching:

Look at index = h'(key). If element not found at index, sequentially look into next indices until

- you find the element
- or reach an index which is NIL (which implies that the element is not in the hash table)

Worst-case Search Complexity?

- In general, for probing based tables, we resize (double) the table size once the **load factor** exceeds a certain value (different for different hashing schemes)
- Load factor = number of elements in table / size of table
- This means the table is never full
- Still O(M) for search, since the table could have load factor * M elements and they could all be in one cluster

Linear Probing

Deletion:

- Search the element
- Delete it
- Is that all?

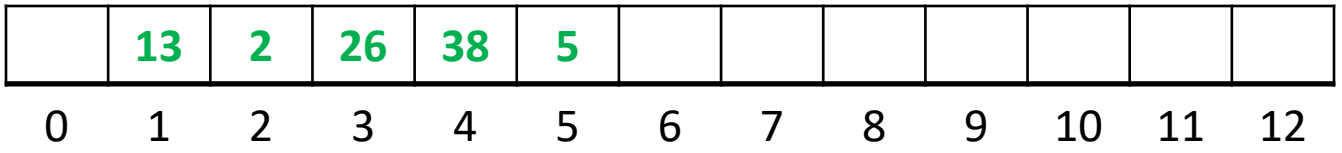
Linear Probing

Deletion:

- Search the element
- Delete it
- Is that all?

Example: Suppose hash function is just **h(k,i) = k%12 + i**

- Delete 2



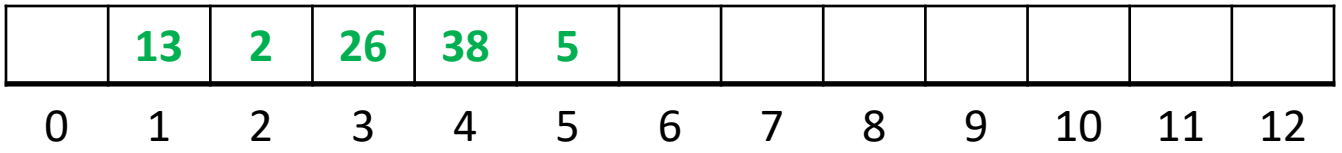
Linear Probing

Deletion:

- Search the element
- Delete it
- Is that all?

Example: Suppose hash function is just **h(k,i) = k%12 + i**

- Delete 2
- 2 is located at position hash(2), so no need to probe



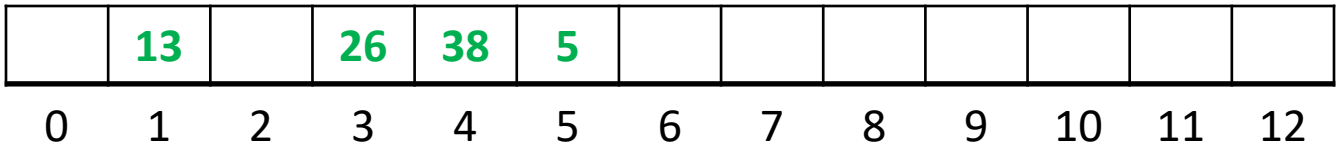
Linear Probing

Deletion:

- Search the element
- Delete it
- Is that all?

Example: Suppose hash function is just **h(k,i) = k%12 + i**

- Delete 2
- 2 is located at position hash(2), so no need to probe
- Delete it



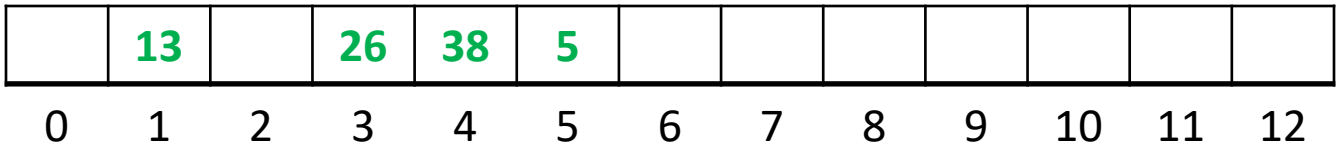
Linear Probing

Deletion:

- Search the element
- Delete it
- Is that all?

Example: Suppose hash function is just **h(k,i) = k%12 + i**

- Delete 2
- 2 is located at position hash(2), so no need to probe
- Delete it
- Now what happens if we look up 38?



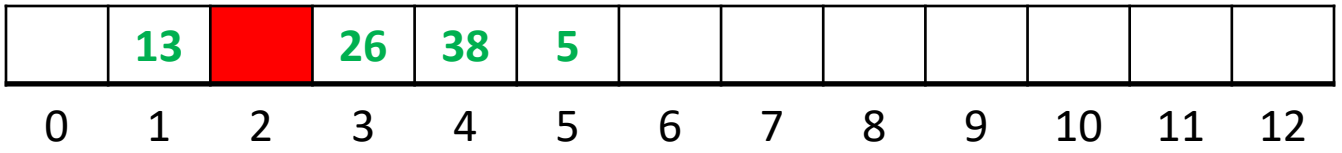
Linear Probing

Deletion:

- Search the element
- Delete it
- Is that all?

Example: Suppose hash function is just **h(k,i) = k%12 + i**

- Delete 2
- 2 is located at position hash(2), so no need to probe
- Delete it
- Now what happens if we look up 38?
- Hash(38)=2, not found. But that is wrong!



Linear Probing

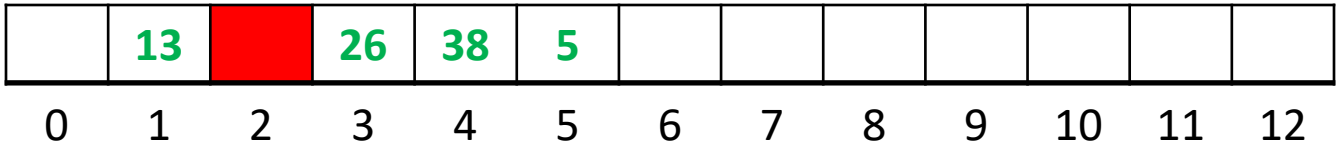
Deletion:

- Search the element
- Delete it
- Is that all?

Example: Suppose hash function is just **h(k,i) = k%12 + i**

- Delete 2
- 2 is located at position hash(2), so no need to probe
- Delete it
- Now what happens if we look up 38?
- Hash(38)=2, not found. But that is wrong!

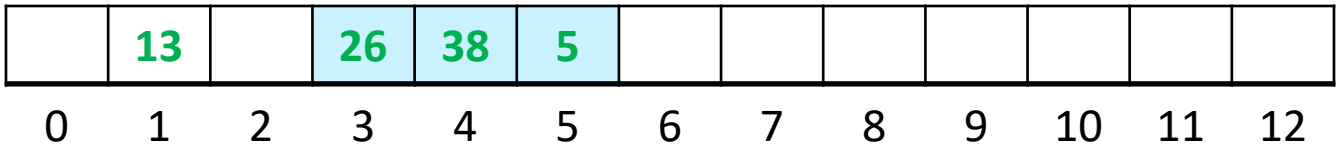
Quiz time!
https://flux.qa - RFIBMB



Linear Probing

Deletion:

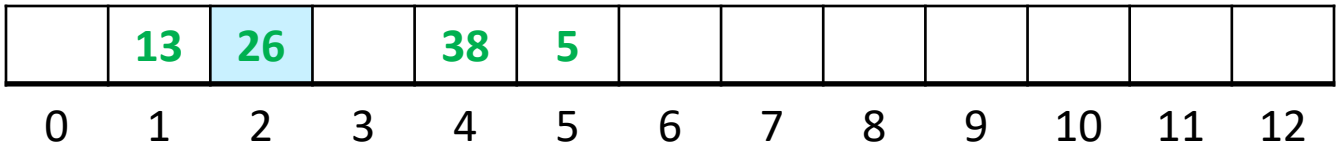
- Search the element
- Delete it
- Is that all?
- Solution: Re-insert everything after the deleted position, until the next empty position



Linear Probing

Deletion:

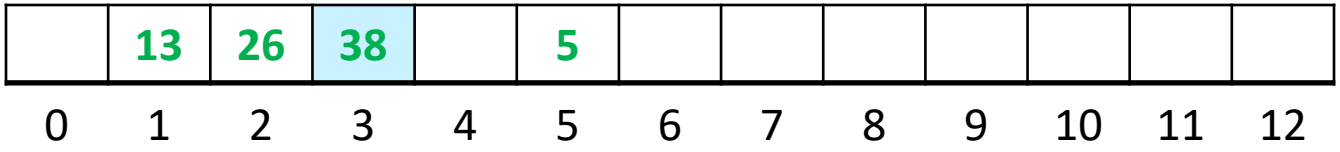
- Search the element
- Delete it
- Is that all?
- Solution: Re-insert everything after the deleted position, until the next empty position
- Insert 26



Linear Probing

Deletion:

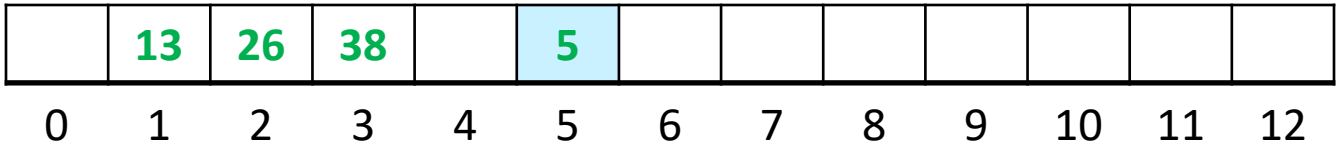
- Search the element
- Delete it
- Is that all?
- Solution: Re-insert everything after the deleted position, until the next empty position
- Insert 26
- Insert 38



Linear Probing

Deletion:

- Search the element
- Delete it
- Is that all?
- Solution: Re-insert everything after the deleted position, until the next empty position
- Insert 26
- Insert 38
- Insert 5



Linear Probing

- The previous example showed a search by increment of 1
- We can increment by any constant: h(k, i) = (h'(k) + **c***i) % M
- E.g., if c=3 and index = 2 is a collision, we will look at index 5, and then index 8, then 11 and so on …

The problem with linear probing is that collisions from nearby hash values tend to merge into big blocks, and therefore the lookup can degenerate into a linear O(N) search. This is called primary clustering.

	13	53	15	30	44	40				23		
0	1	2	3	4	5	6	7	8	9	10	11	12

Outline

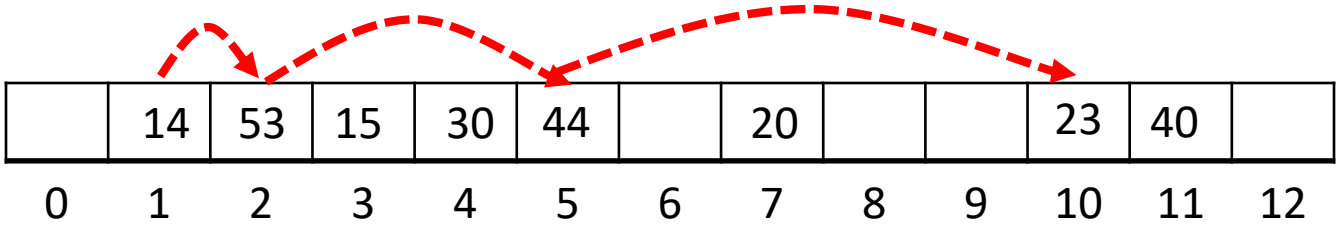
1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables
   A. Introduction
   B. Chaining
   C. Probing
      I. Linear Probing
      II. Quadratic Probing
      III. Double Hashing
      IV. Cuckoo Hashing

Quadratic Probing

Unlike linear probing that uses fixed incremental jumps, quadratic probing uses quadratic jumps.

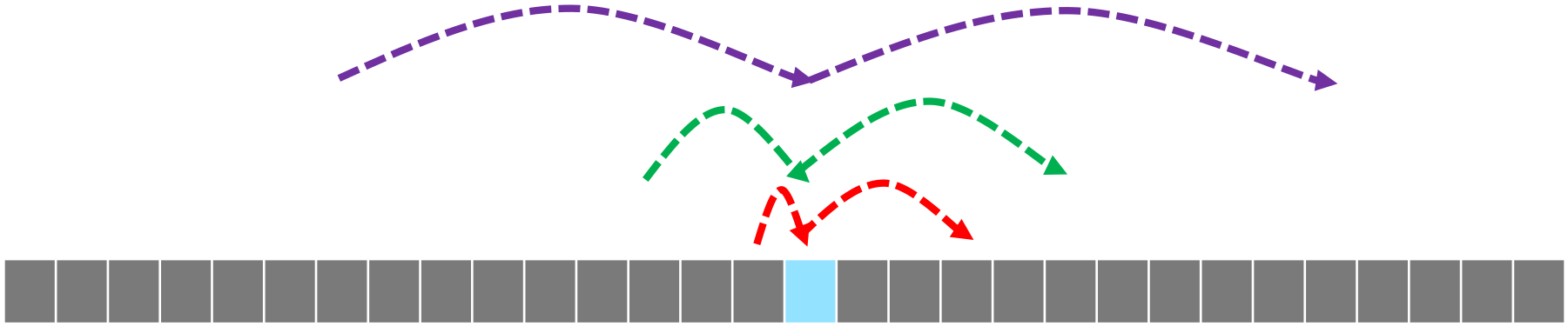
- Linear probing: h(k, i) = (h'(k) + c*i) % M
- Quadratic probing: h(k, i) = (h'(k) + c*i + d*i^2) % M where c and d are constants

- Quadratic probing is not guaranteed to probe every location in the table.
  - An insert could fail while there is still an empty location
  - It can be shown that with prime table size and load factor < 0.5, quadratic probing always finds an empty spot
  - Therefore, make sure we resize when load factor = 0.5!
  - The same probing strategy is used in the associated search and delete routine!



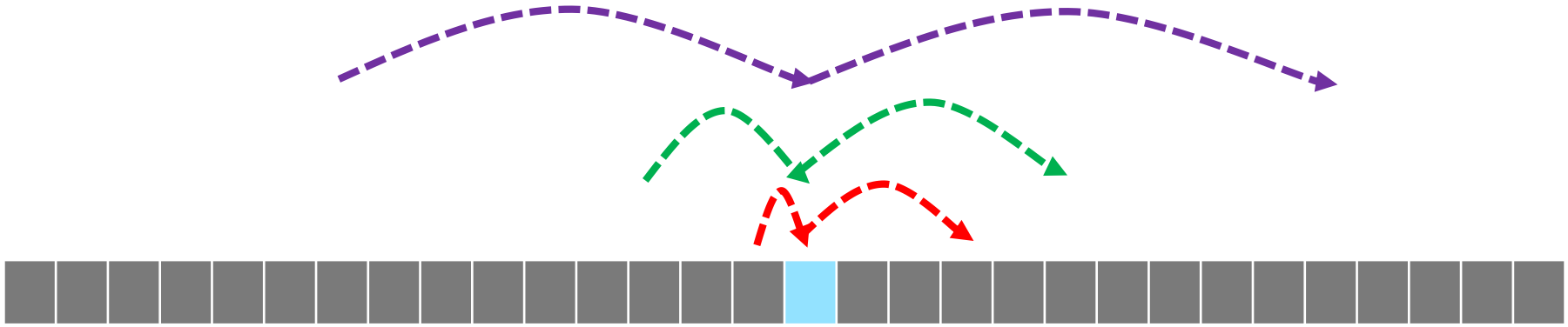
Quadratic Probing - Delete

- In linear probing, each element **k** may be part of a probe chain, but the previous position in the chain will be 1 (or c) positions before **k,** and the next position in the chain will be 1 (or c) positions after **k**
- For quadratic probing, an element may be part of many, separate probe chains
- Which elements do we re-insert?



Quadratic Probing - Delete

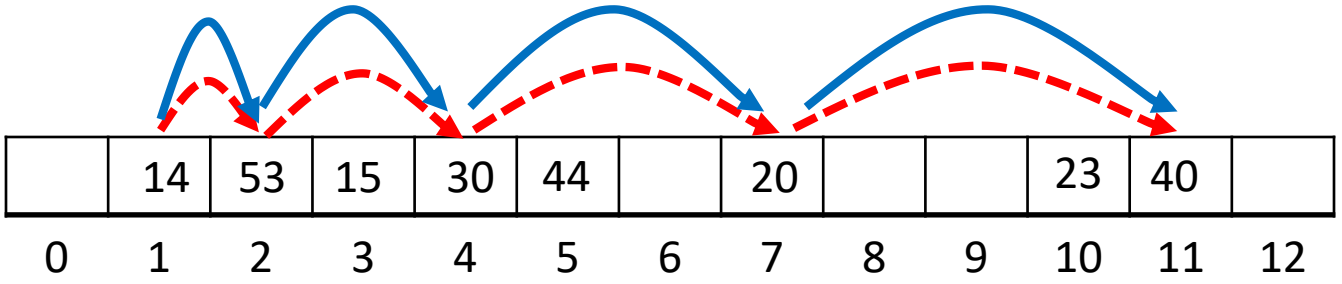
- Instead of reinserting, we use “lazy deletion”
- Simply **mark** the element as “deleted”
- When we do lookups, we do not return **marked** elements (but we can probe past them)
- When we do inserts, we are allowed to insert over the top of **marked** elements



Problem with Quadratic Probing

- Quadratic probing avoids primary clustering
- However, if two elements have same hash index (e.g., h'(k1) = h'(k2)), the jumps are the same for both elements.

- This leads to a milder form of clustering called secondary clustering.
- Is there a way to have different "jumping" for elements that hash to same indexing?
  - Yes! Double hashing – two hash functions
  - Or Cuckoo hashing – two hash functions and two hash tables

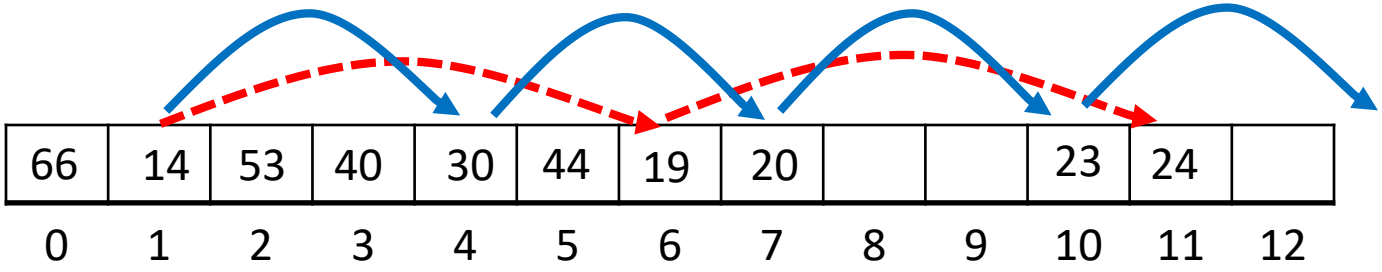


Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables
   A. Introduction
   B. Chaining
   C. Probing
      I. Linear Probing
      II. Quadratic Probing
      III. Double Hashing
      IV. Cuckoo Hashing

Double hashing

- Use two different hash functions: one to determine the initial index and the other two determine the amount of jump
- h(k, i) = (hash1(key) + i*hash2(key)) % M
- E.g., suppose hash1() is key % 13 and hash2() is key % 7
- Insert 40 → hash1(40) = 1, hash2(40) = 5
  - i = 0 → index = 1
  - i = 1 → index = (1+5)%13 = 6
  - i = 2 → index = (1+10)%13 = 11
  - i = 3 → index = (1+ 15)%13 = 3
- Insert 66 → hash1(66) = 1, hash2(66) = 3
  - i = 0 → index = 1
  - i = 1 → index = (1+3)%13 = 4
  - i = 2 → index = (1+6)%13 = 7
  - i = 3 → index = (1+ 9)%13 = 10
  - I = 4 → index = (1+ 12)%13 = 0



Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables
   A. Introduction
   B. Chaining
   C. Probing
      I. Linear Probing
      II. Quadratic Probing
      III. Double Hashing
      IV. Cuckoo Hashing

Cuckoo hashing

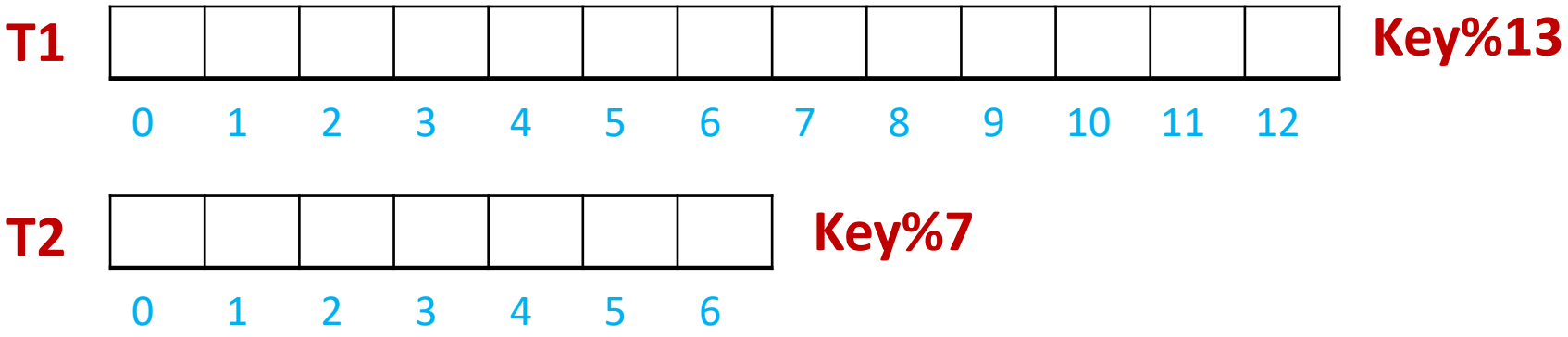
None of the hashing schemes seen earlier can provide guarantee for O(1) searching in the worst-case.

Cuckoo Hashing:

- Searching – O(1) in worst-case
- Deletion – O(1) in worst-case
- Insertion cost may be significantly higher – but average cost is O(1)

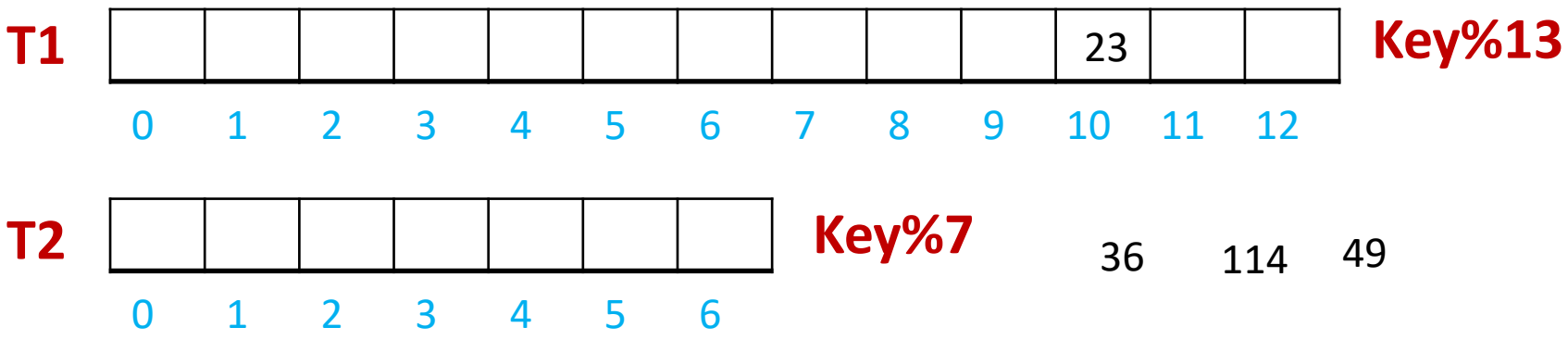
Idea:

- Use two different hash functions hash1() and hash2() and two different hash tables T1 and T2 of possibly different sizes.
- hash1() determines the index in T1 and hash2() determines the index in T2.
- Each key will be indexed in only one of the two tables
- Handle collision by "Cuckoo approach"
  - kick the other key out to the other table until every key has its own "nest" (i.e., table)



Cuckoo hashing

- Insert 23
  - Hash1(23) → 23%13 →10 — Insert 23 at T1[10]
- Insert 36
  - Hash1(36) → 36%13 → 10 — Insert 36 at T1[10] and kick away 23 to T2
  - Hash2(23) → 23%7 →2 — Insert 23 at T2[2]
- Insert 114
  - Hash1(114) → 114%13 → 10 — Insert 114 at T1[10] and kick away 36 to T2
  - Hash2(36) → 36%7 → 1 — Insert 36 at T2[1]
- Insert 49
  - Hash1(49) → 49%13 → 10 — Insert 49 at T1[10] and kick away 114 to T2
  - Hash2(114) → 114%7 → 2 — Insert 114 at T2[2] and kick away 23 to T1
  - Hash1(23) → 23%13 → 10 — Insert 23 at T1[10] and kick away 49 to T2
  - Hash2(49) → 49%7 → 0 — Insert 49 at T2[0]



Cuckoo hashing

- Insert 140
  - Hash1(140) → 140%13 → 10
  - Hash2(23) → 23%7 → 2
  - Hash1(114) → 114%13 → 10
  - Hash2(140) → 140%7 → 0
  - Hash1(49) → 49%13 → 10
  - Hash2(114) → 114%7 → 2
  - …

- Cuckoo Hashing gives up after **MAXLOOP** number of iterations
- Uses new hash functions (and may resize two tables) and hashes everything again
  - Thus insertion may be quite costly

Insert 140 at T1[10] and kick away 23 to T2

Insert 23 at T2[2] and kick away 114 to T1

Insert 114 at T1[10] and kick away 140 to T2

T1

0	1	2	3	4	5	6	7	8	9	10	11	12
										23		

Key%13

T2

0	1	2	3	4	5	6
49	36	114				

Key%7

140

Cuckoo hashing

Observation: A **key** is either at index= hash1(key) in T1 or at index = hash2(key) in T2

Searching:

- Look at T1[hash1(key)] and T2[hash2(key)]

E.g.,

Search 36

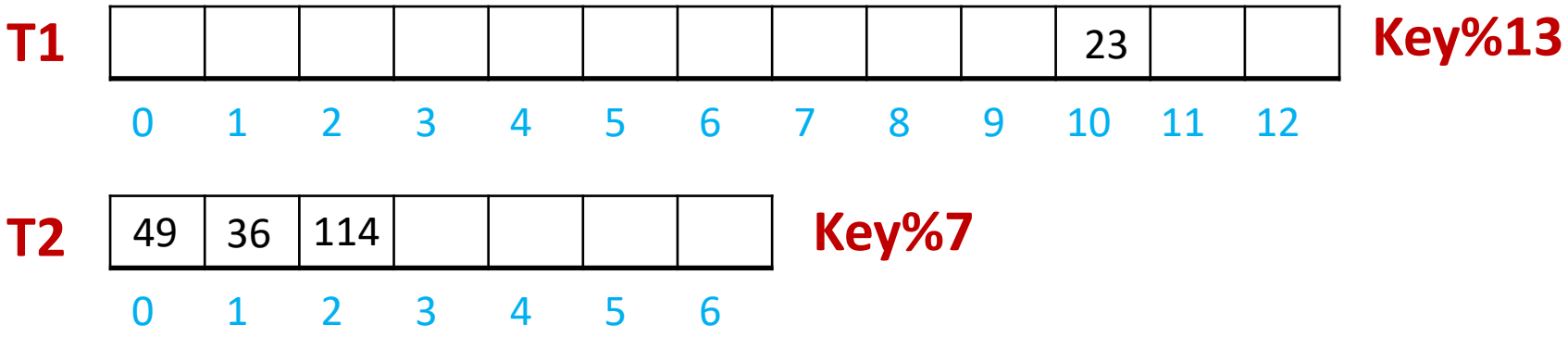
- 36%13 →10 Look at T1[10]. Not there, so look in T2
- 36%7 → 1 Look at T2[1]. Found!!!

Search 10

- 10%13 →10 Look at T1[10]. Not there, so look in T2
- 10%7 → 3 Look at T2[3]. Not found!!!

What is worst-case time complexity for searching?

- **O(1)**



Cuckoo hashing

Observation: A **key** is either at index= hash1(key) in T1 or at index = hash2(key) in T2

Deletion:

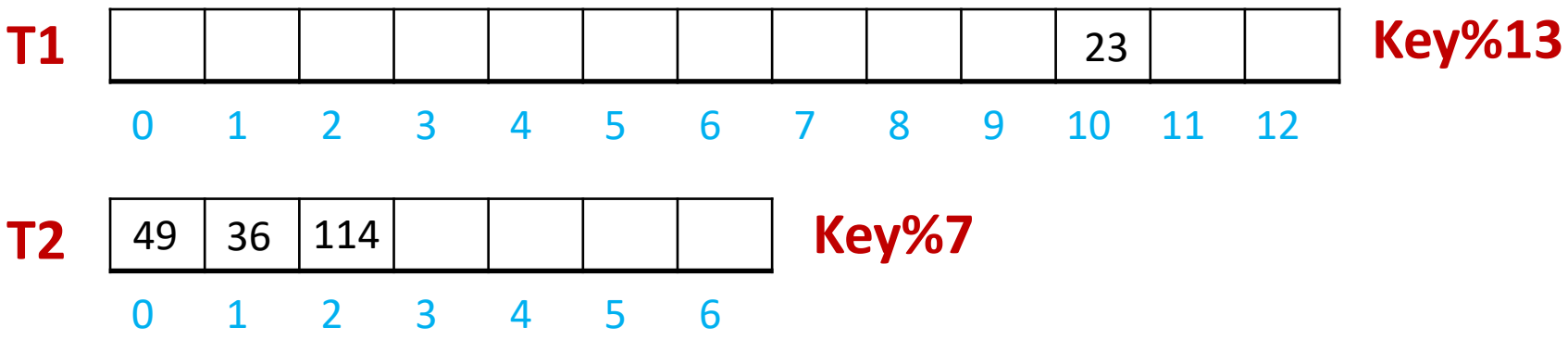
- Search key
- Delete it if found

What is worst-case time complexity for deletion?

- **O(1)**

What about insertion? (NON-EXAMINABLE)

- **In the worst-case, it could be arbitrarily bad**
- **It has been shown that the expected cost to insert N items is O(N)**



Tables setups vs better hashing

- So far we have talked about ways to arrange our table
- We can also have better hash functions!

- The game:
  - You make a hash table algorithm
  - I (an evil person) pick some inputs
  - You make the table and put the inputs into your table

- If you choose any deterministic hash, I can break it as follows:

Tables setups vs better hashing

- Suppose h(k) is your hash function
- I choose values x_i, x_2 … x_n, such that $h(x_i) = c$ for all i
- All x_i will collide

- If you try to fight this by making a random hash function, I cannot reverse engineer it in this way
- **But** you will be unable to find items (since when you hash them again, you will end up with a different index)
- Since you have to pick your hash function when you make the table, there is seemingly no way around this

Tables setups vs better hashing

- Solution: Use a random **family** of hash functions

- Whenever you make a table, choose a function at random from an infinite family
- Example: **a**k + **b** mod **p**
- **p** is a prime larger than table size
- **a** and **b** are chosen randomly, but
- **a** can't be 0 mod p (a multiple of p)

Tables setups vs better hashing

- Universal family: for any two keys, the chance of them being hashed to the same position by a randomly chosen function from the universal family is at most 1/tablesize

$$\Pr_{h \in \mathcal{H}} \{h(k) = h(k')\} \leq \frac{1}{m}, \qquad \text{for all } k \neq k' \in \mathcal{U}$$

Tables setups vs better hashing

- There are better properties than just **universality,** like k-independence, but they are beyond the scope of this unit
- For some hashing schemes, we need certain mathematical properties of our hash functions to guarantee our time complexities

Summary of Hashing

- It is hard to design good hash functions
- Imitating randomness is the ultimate goal of a good hash function
  - But it still needs to be deterministic!
- The examples shown in the lecture show very simple hash functions (see notes!)
- In practice hash tables give quite good performance, i.e. O(1) on average
- Hash tables are disordered data structures. Therefore certain operations become expensive.
  - Find maximum and minimum of a set of elements (keys).

Summary

Take home message

- Hash tables provide O(1) look up in practice (although the worst-case complexity may still be O(N))
- AVL Trees guarantee worst-case time complexity of O(Log N)

Things to do (this list is not exhaustive)

- Read more about hash tables and hash functions
- Practice balancing AVL trees using pen and paper
- Implement BST and AVL trees

Coming Up Next

- Retrieval Data Structures for Strings