Faculty of Information Technology,
Monash University

# FIT2004: Algorithms and Data Structures

## Week 5: Efficient Lookup Structures

These slides include materials prepared by M. A. Cheema, Arun Koagurthu and Lloyd Allison.

# Recommended Reading

- Unit Notes – chapters 7 and 8

# Outline

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

# Lookup Table

- Key Value pairs, eg. Name – birthday
- 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
- **Is the Monash database just a big Lookup Table?**

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

# 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

# Dictionaries in Python

```
grocery_items =
        {"eggs": 3.99,
          "banana": 1.49,
          "cheese": 4.5,
          "eggplant": 2.5,
          "bread": 3.99}


grocery_items["onion"] = 3.50

grocery_items.pop("banana")

grocery_items.keys()
```

# sorting based lookup in Array

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) – remember we work in an Array
  - 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

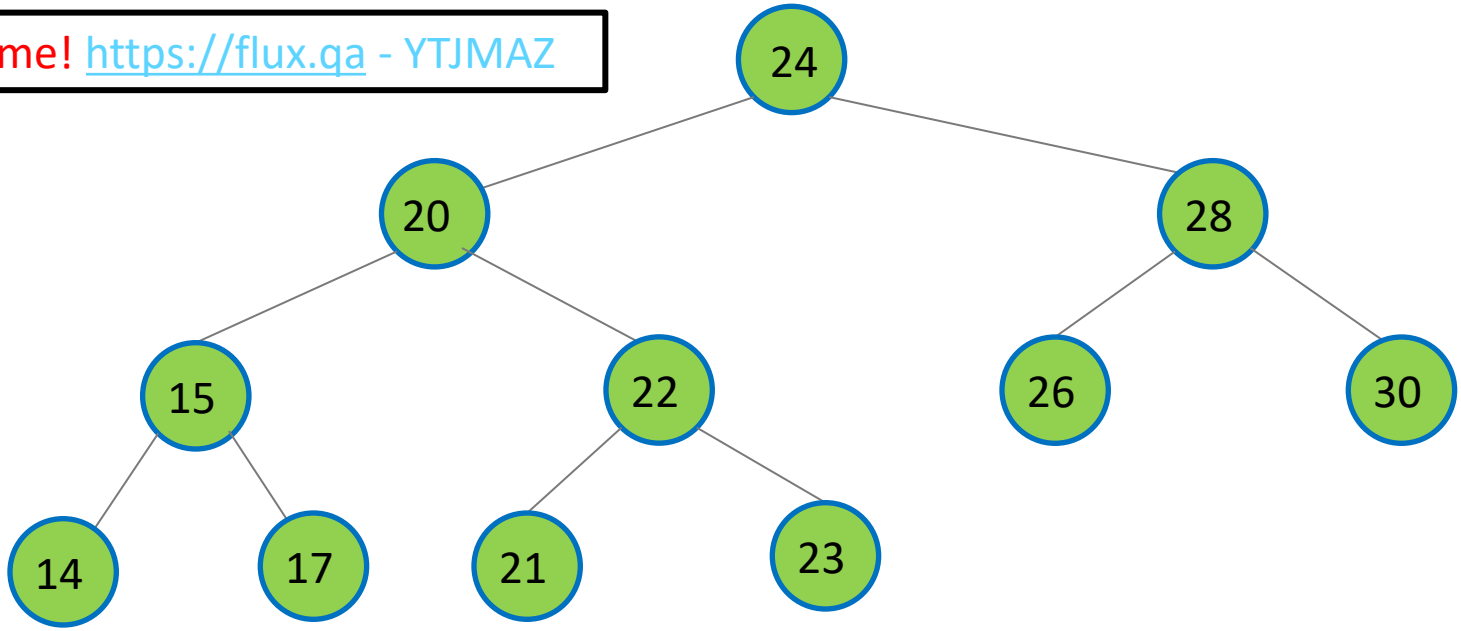
1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables

# 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

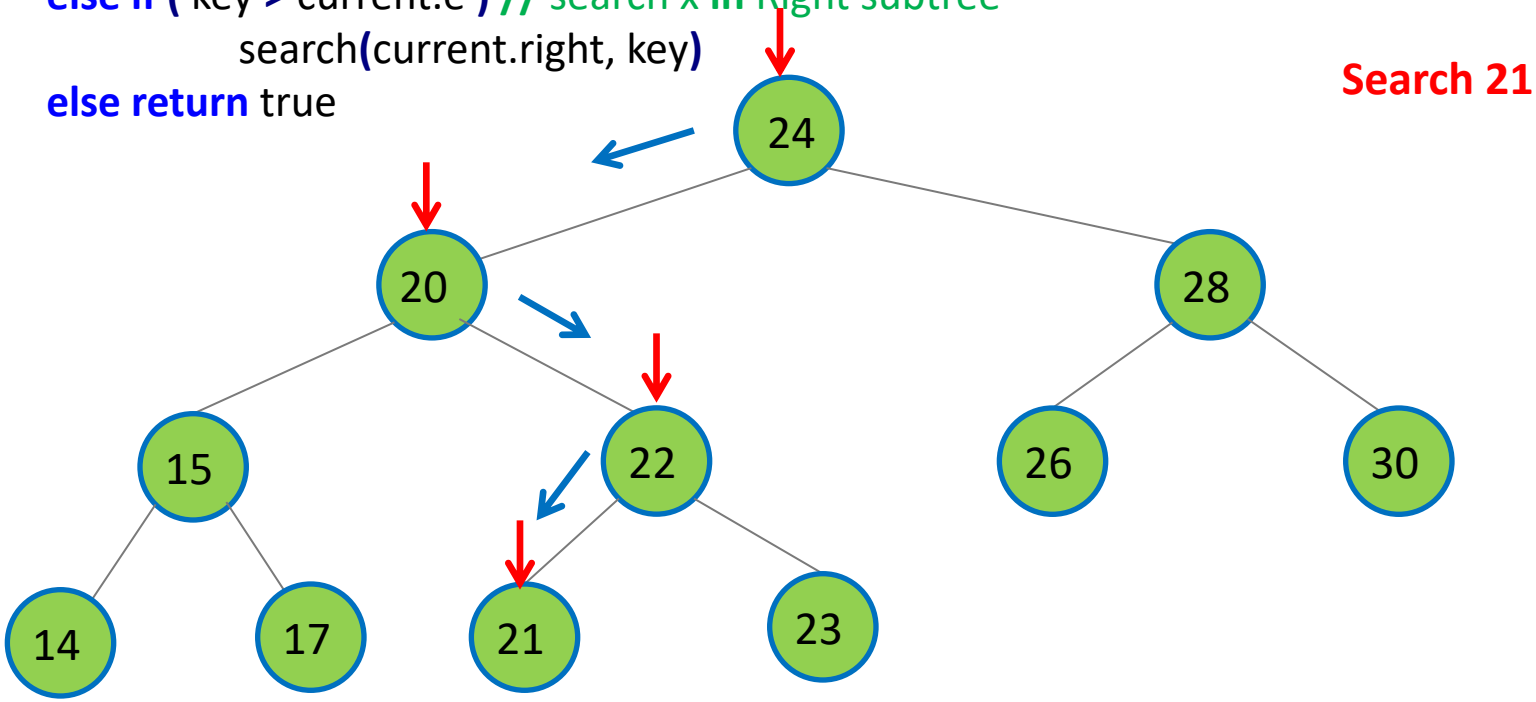
**Anything missing??**

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



# Searching a key in BST

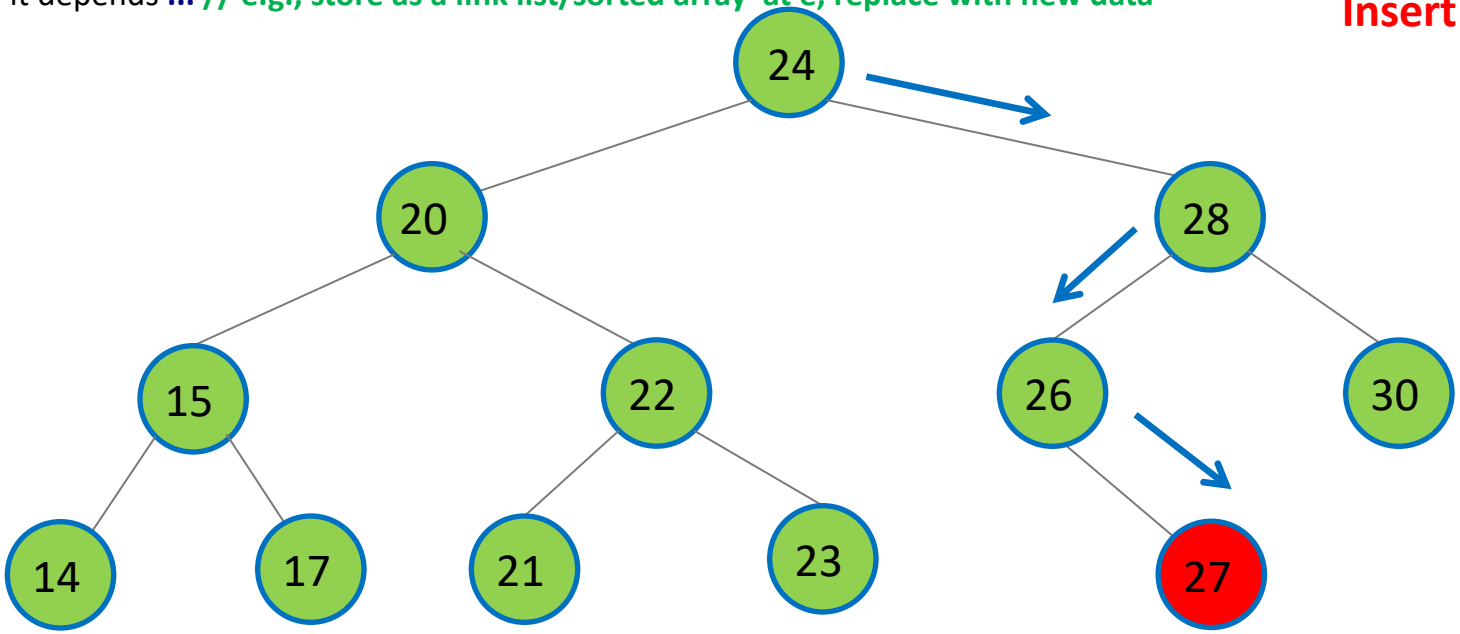
```
// BST implemented here as a tree data structure
def search(current, key)
        if (current == None)
           return false // not present!
        else if ( key < current.e ) // search x in Left subtree
                search(current.left, key)
        else if ( key > current.e ) // search x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



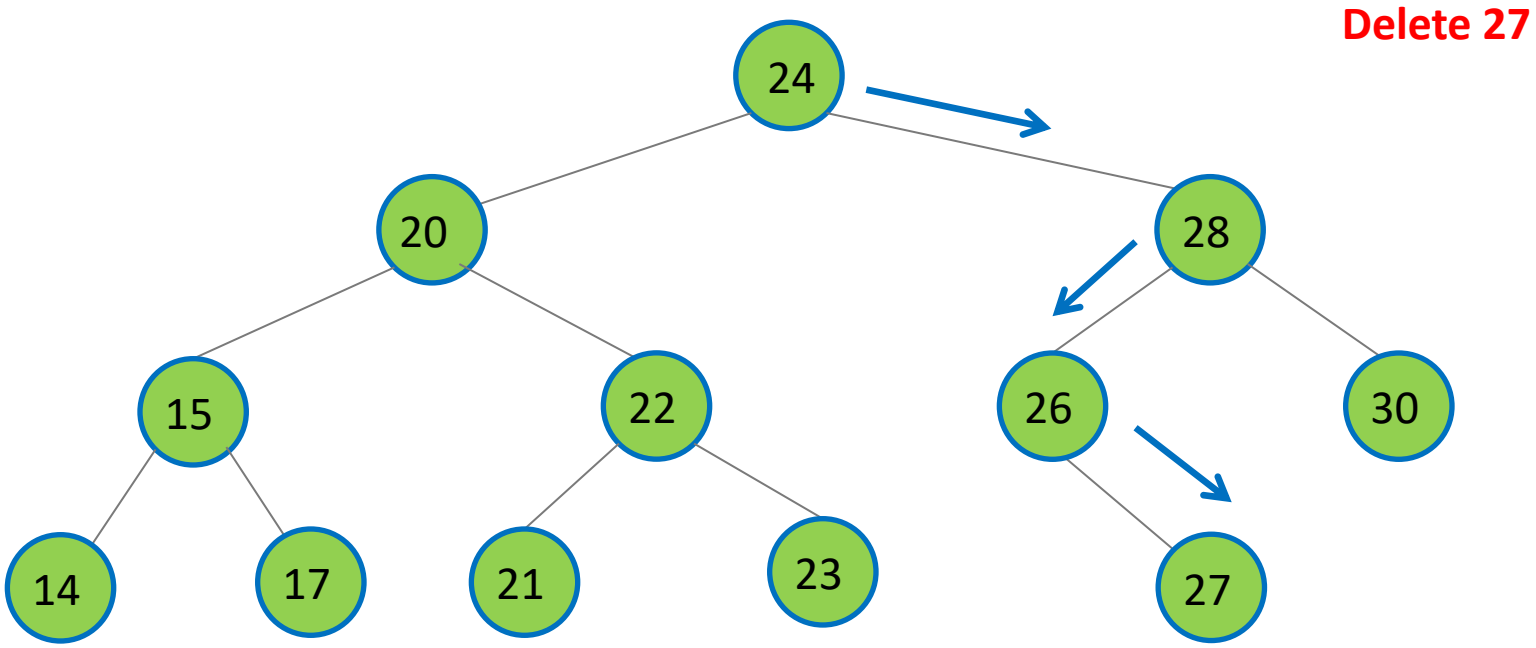
# Delete a key from BST

**First lookup key in BST**

**If the key node is a leaf (has no children) // Case 1**

**delete the key node**

**set subtree to nil**



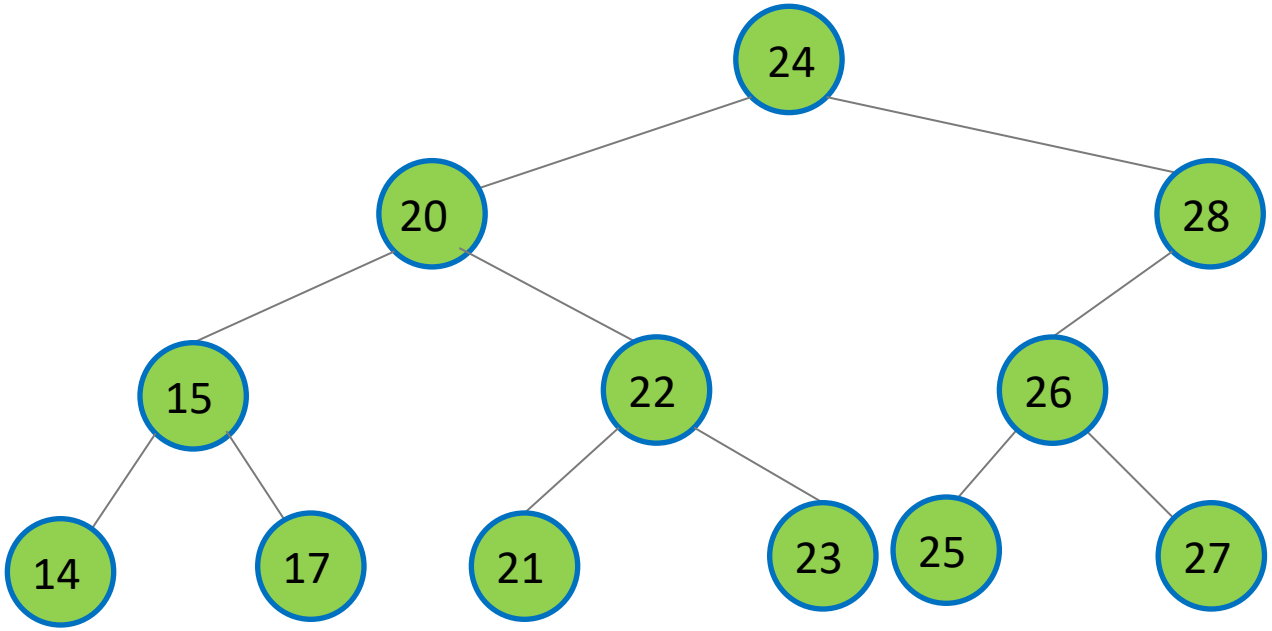
# Delete a key from BST

**First lookup key in BST**

**If the key node has a single child // Case 2**

**delete the key node**

**replace the key node with its child**




**Delete 28**

# Delete a key from BST

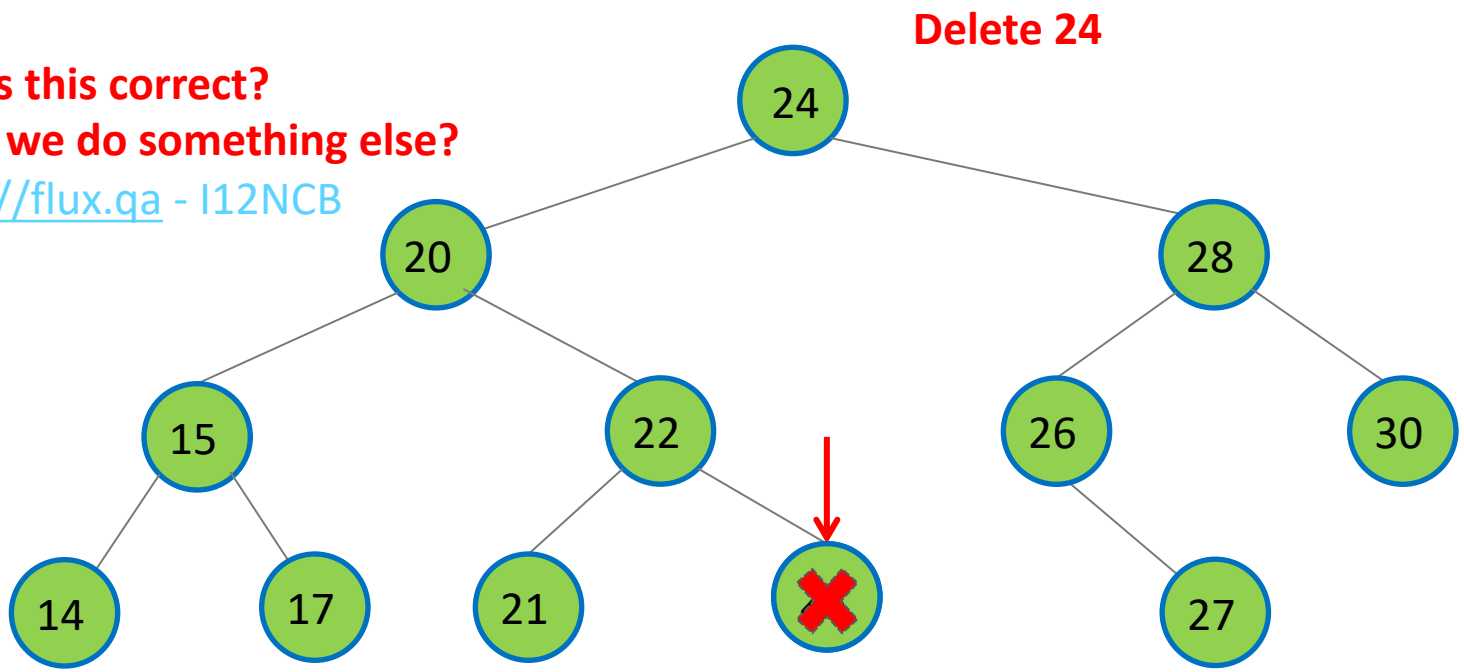
**First lookup key in BST**

**If the key node has two children // Case 3**

**Replace key node with rightmost item in the left subtree**

**Delete this item**

**Why is this correct?**
**Could we do something else?**
https://flux.qa - I12NCB



# Delete a key from BST

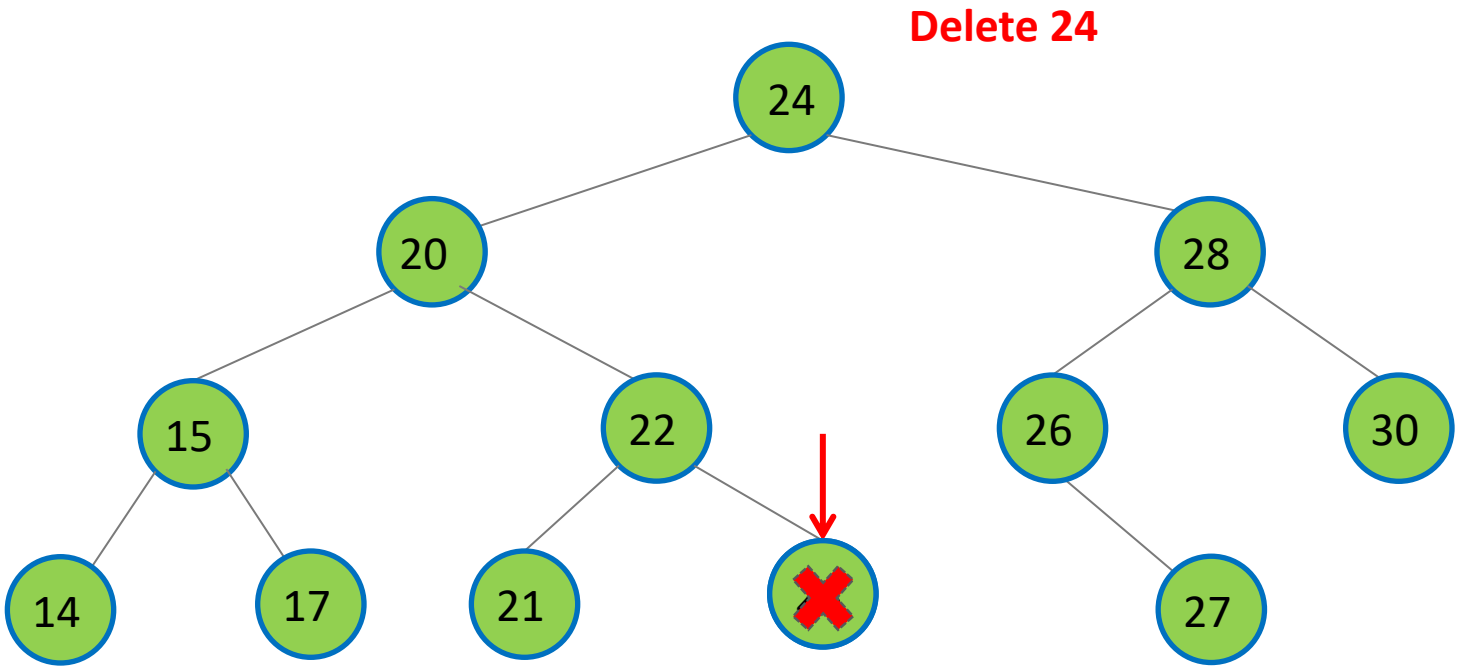
First lookup key in BST

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

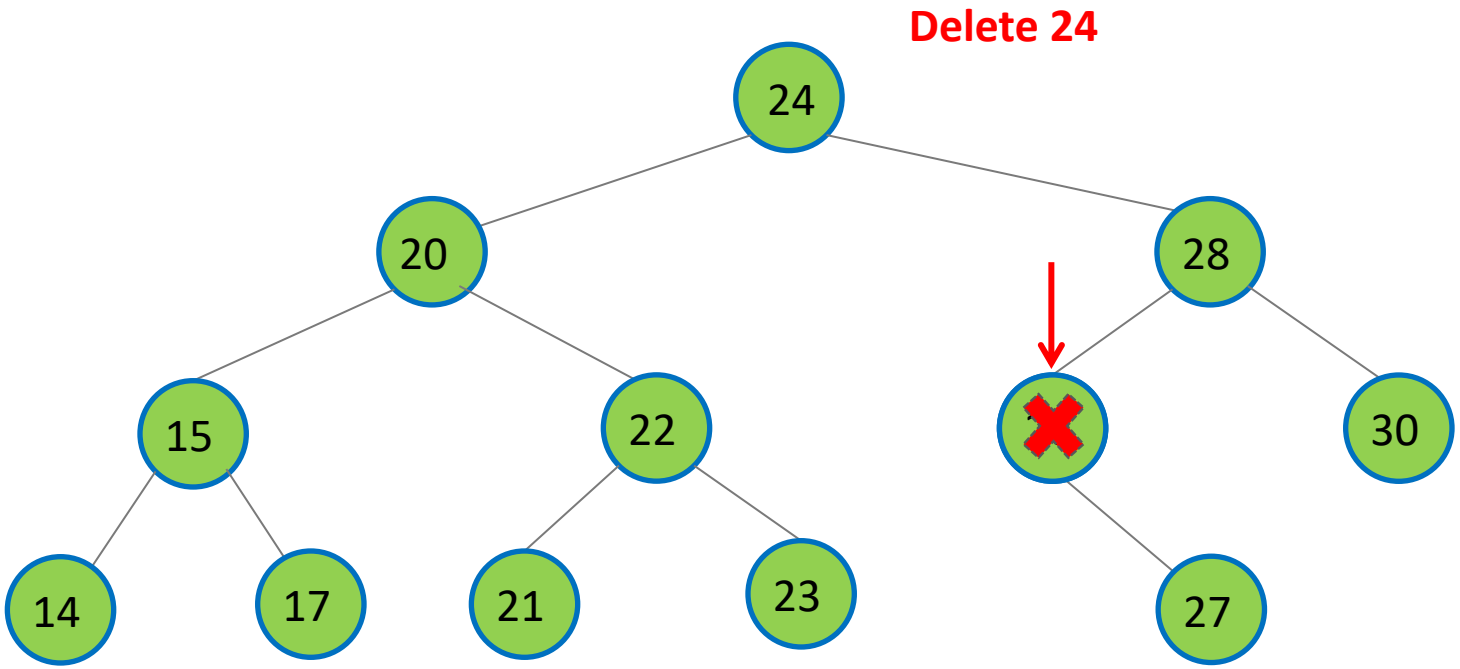
**First lookup key in BST**

**If the key node has two children // Case 3**

**Find successor (or predecessor)**

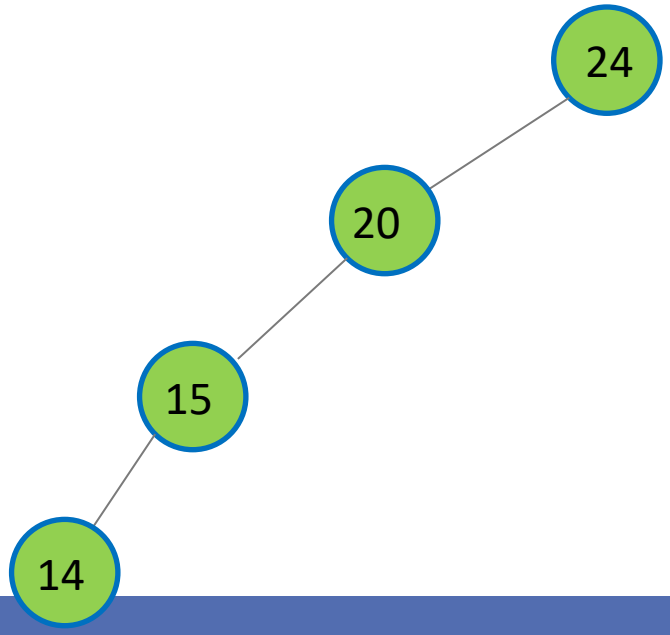
**Replace key node value with the value of successor (or predecessor)**

**Delete successor (or predecessor)**



# Worst-case of BST

- A BST, in worst case, can 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?



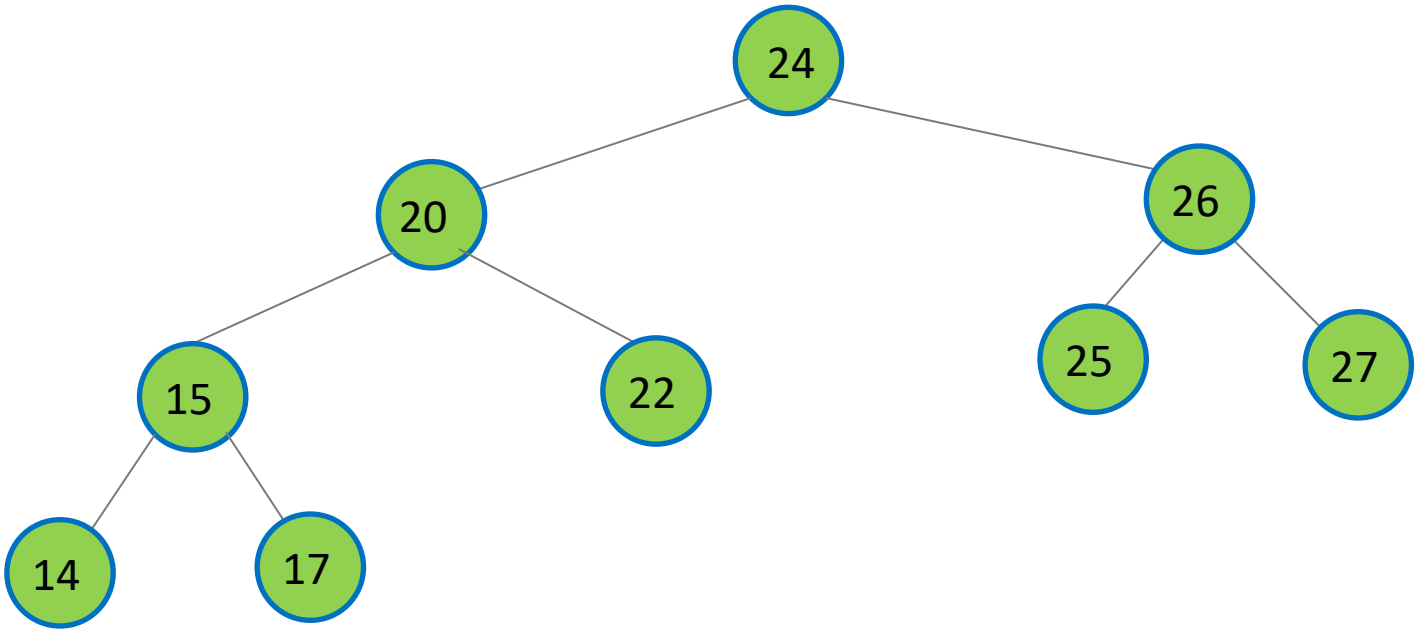
# Outline

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, 1962) 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.

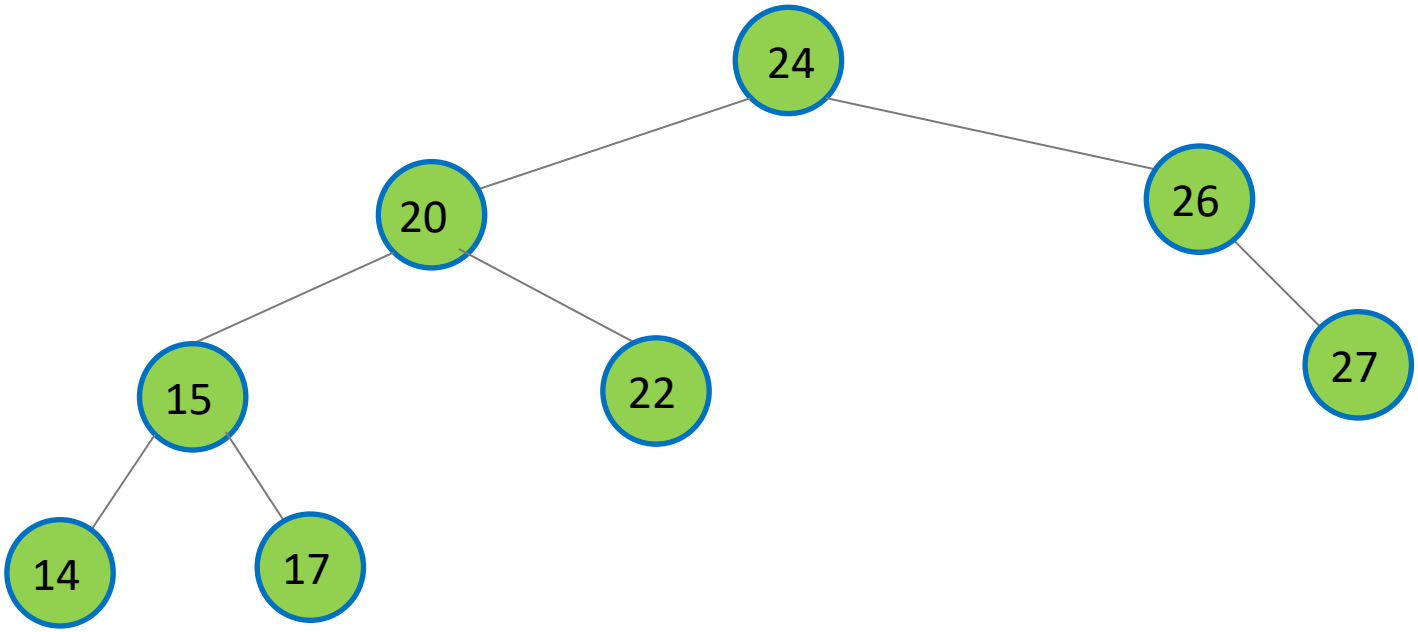
- Is the following tree balanced according to the above definition?



# 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?

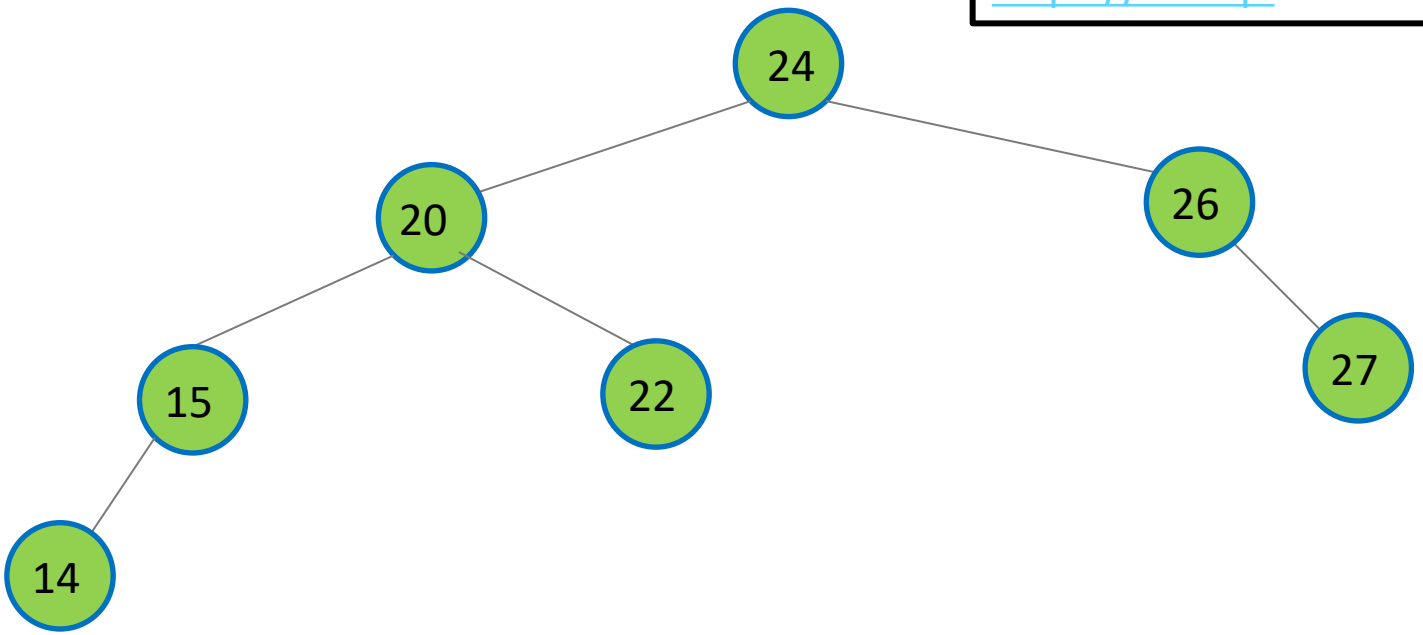


# AVL Trees: Introduction

- 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?

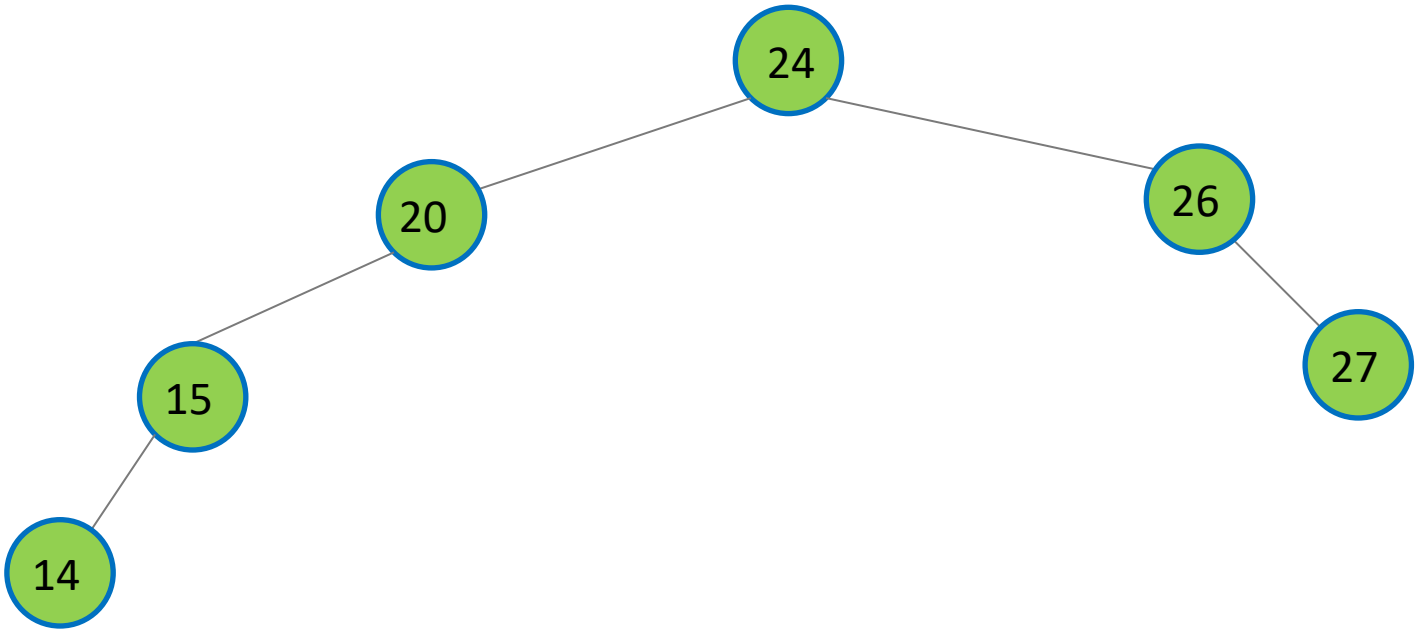
Quiz time!
https://flux.qa - YTJMAZ



# 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?



# AVL Trees: Height

- 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.

- How do we know this is useful?
- Height of an AVL tree with n nodes is O(log n)!

# AVL Trees: Height

- Minimal number of nodes in a tree of depth d is

(1) $T(d) = T(d-1) + T(d-2) + 1$

(2) Let $R(d)=T(d)-T(d-1)$

(3) $R(d)=T(d-2)+1$ rearrange (1) and substitute into (2)

(4) $R(d+2) = T(d) + 1$ from (3)

(5) $R(d+2) = T(d-1)+T(d-2)+1+1$ substitute (1) into (4)

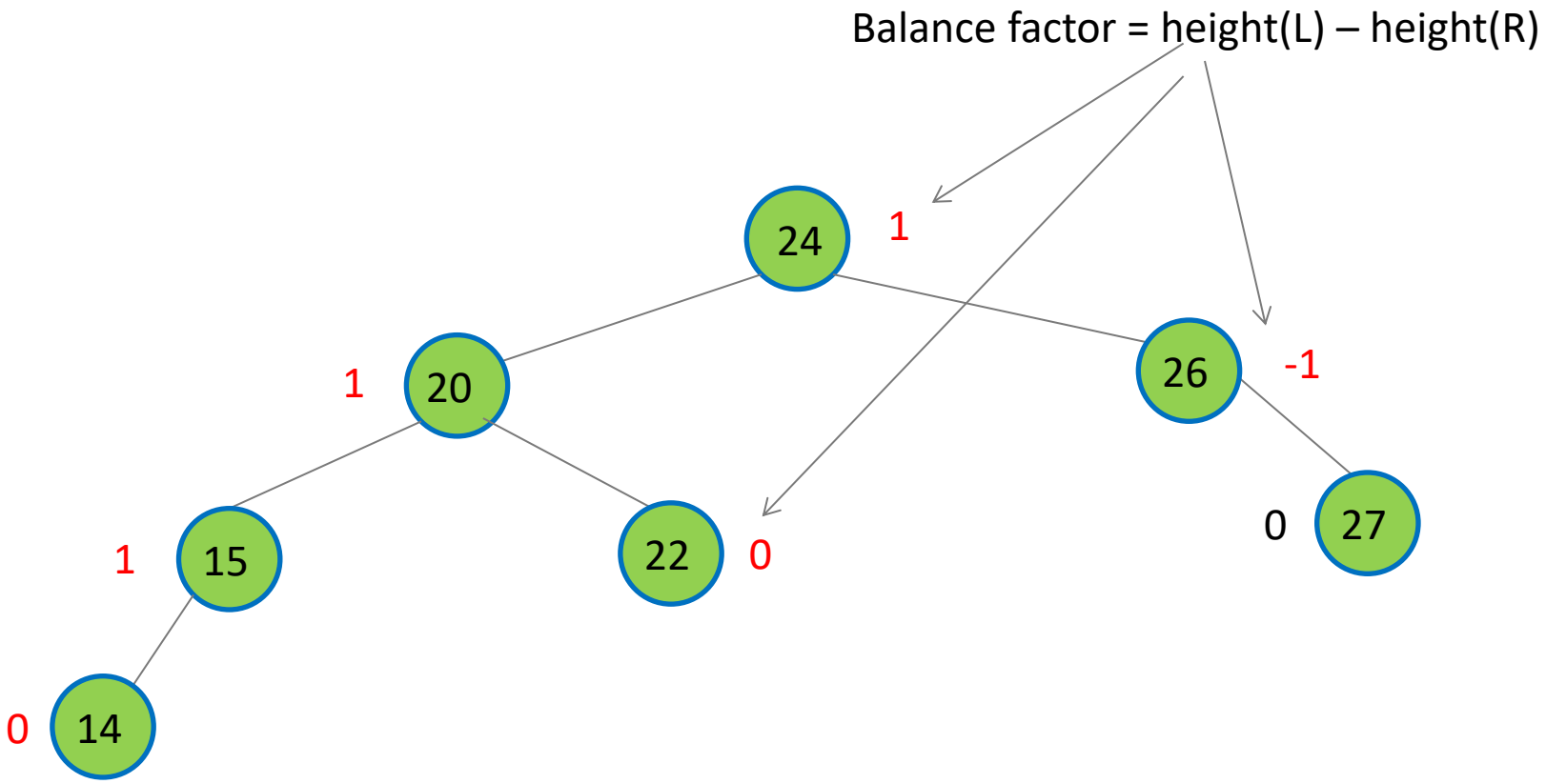
(6) $R(d+2) = R(d) + R(d+1)$ substitute (3) into (5)

- So R has the structure of the Fibonacci Sequence
- R(n)=Fib(n+1) also matches the base cases
- We know that Fib grows exponentially from Binet's formula

- Thus T(d) also grows exponentially (as $\phi^n$)
- Inverting this, we reach d=O(log n)

# 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

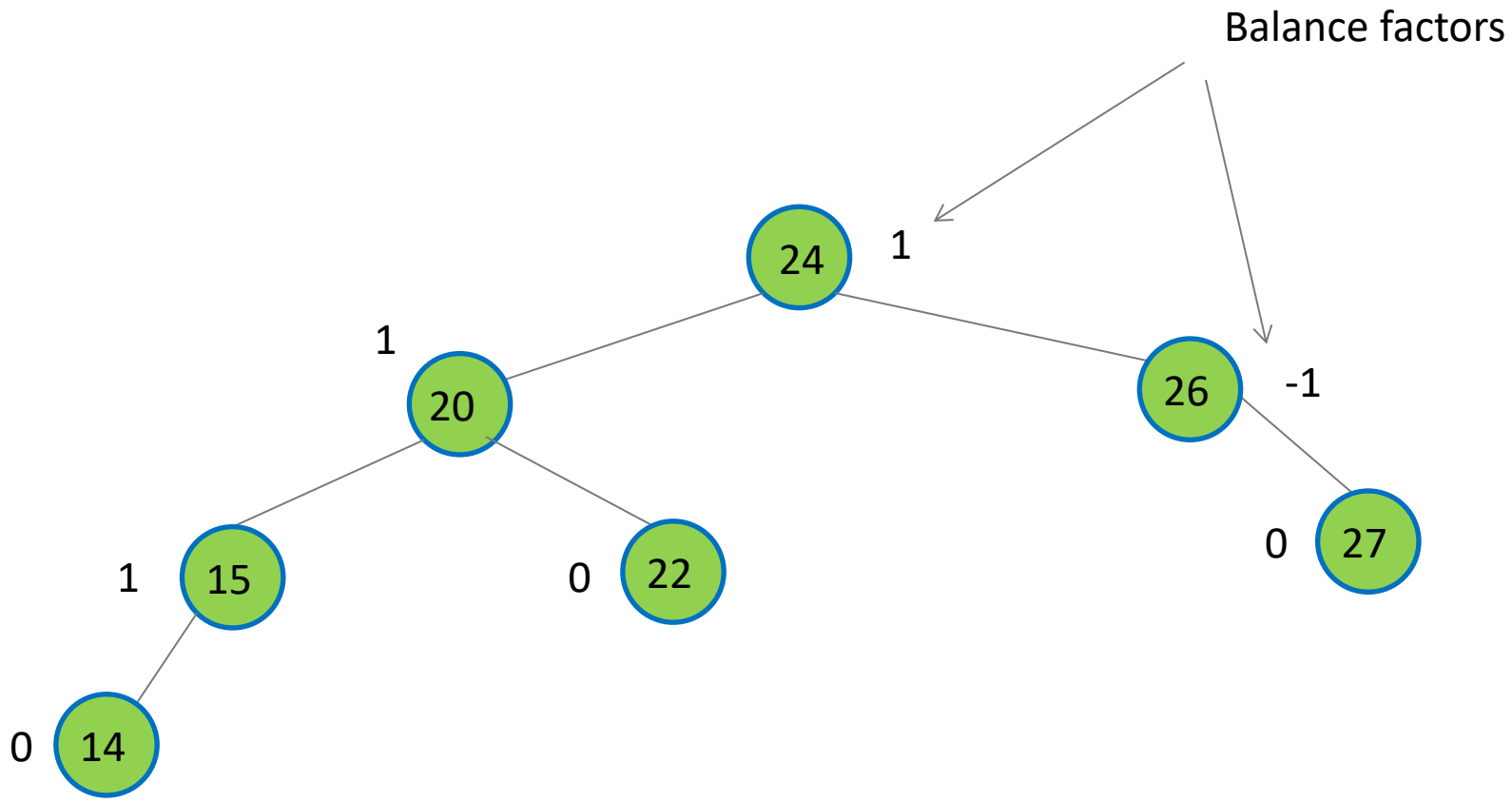


# Outline

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

First insert/delete as in BST, then rebalance

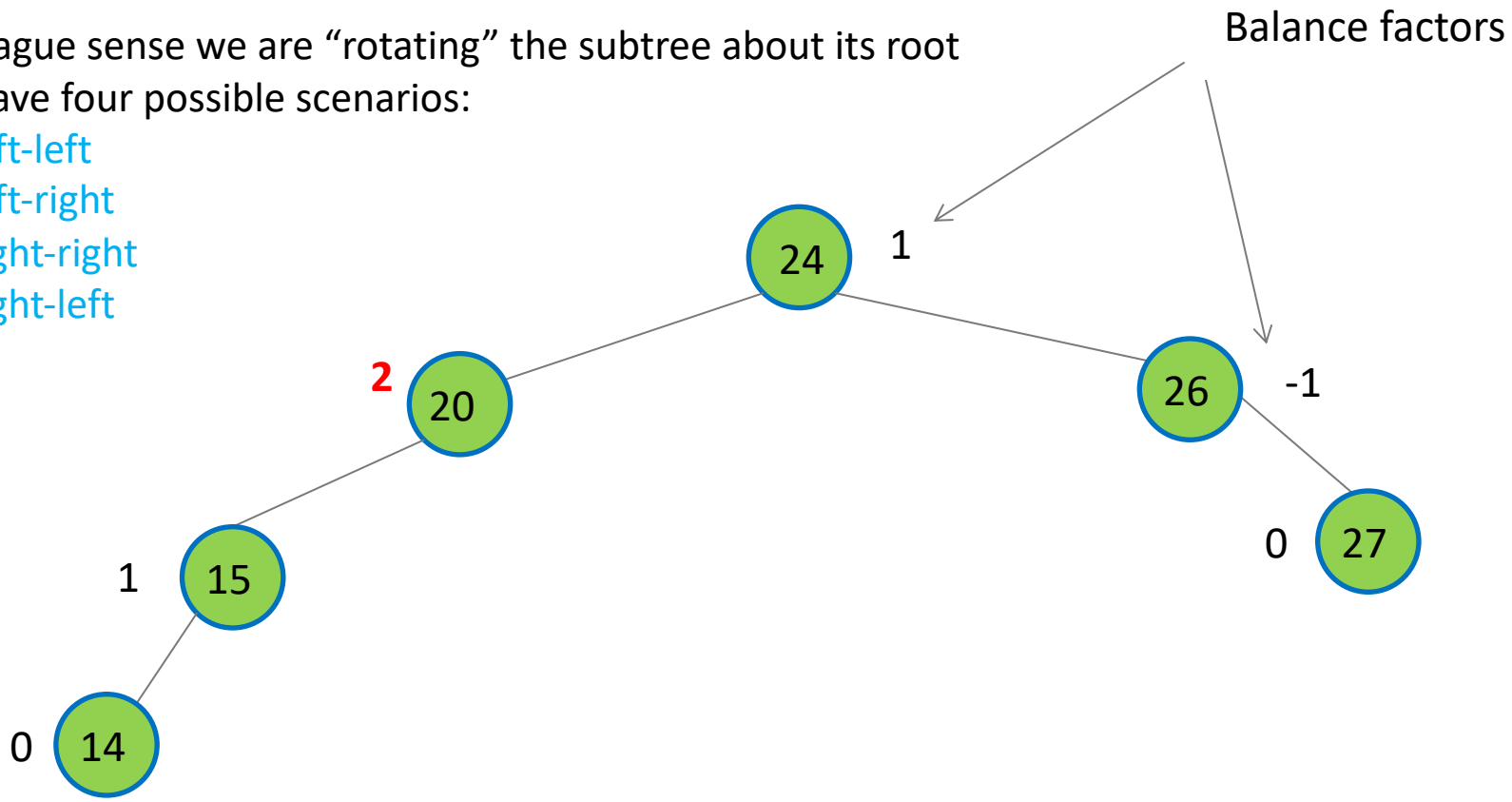
- The tree becomes unbalanced after deleting 22.
- The tree may also become unbalanced after insertion.

How to balance it after deletion/insertion?

In a vague sense we are “rotating” the subtree about its root

We have four possible scenarios:

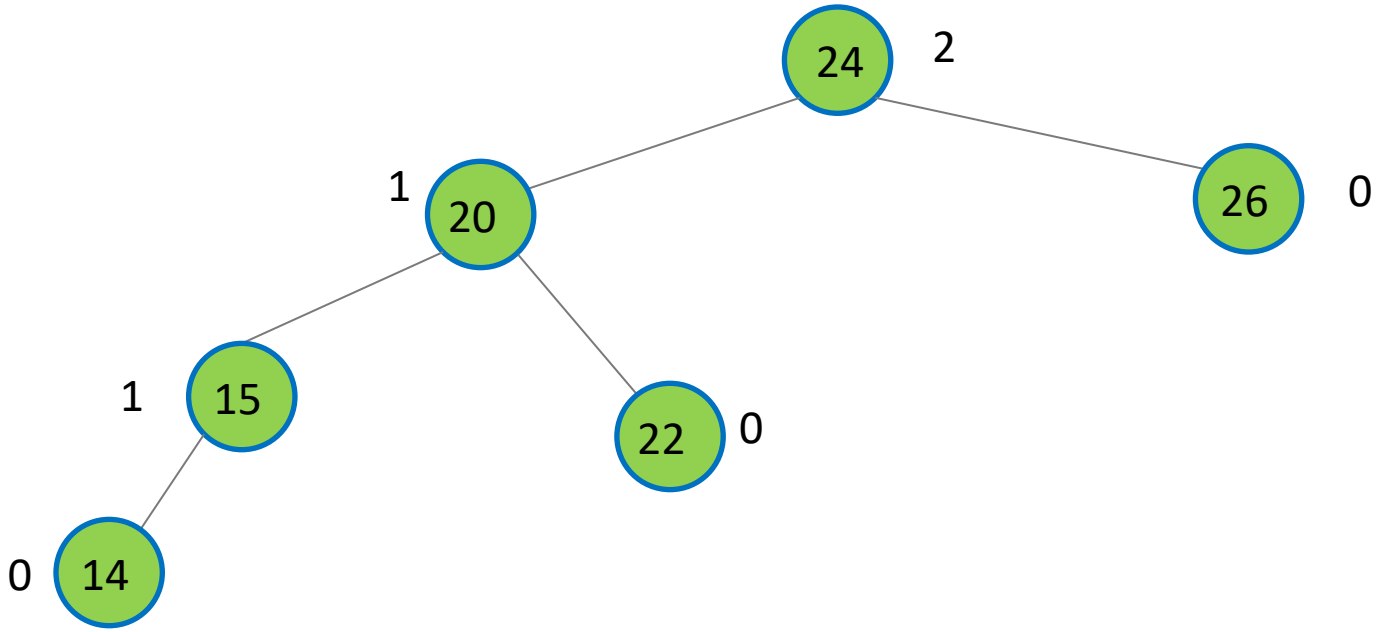
- Left-left
- Left-right
- Right-right
- Right-left



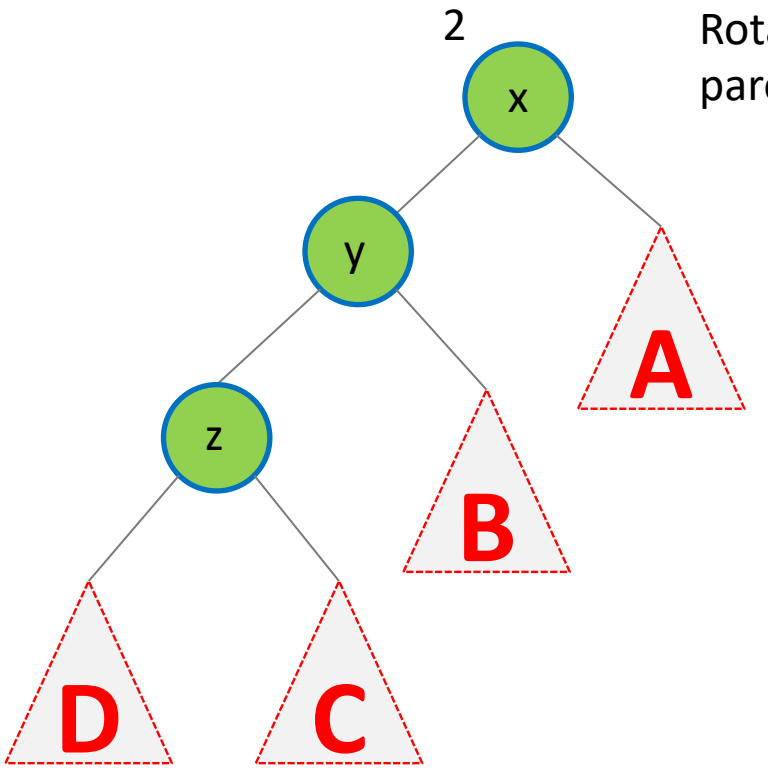
# Left-Left Case

A left-left case occurs when

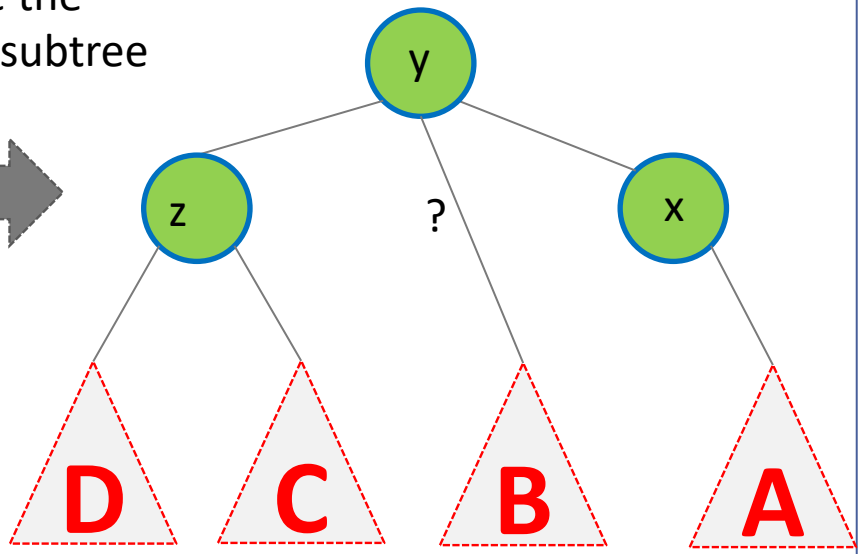
- A node has a balance factor +2; and
- Its left child has a balance factor 0 or more
- i.e the path to the deepest node turns left twice



# Handling Left-left case



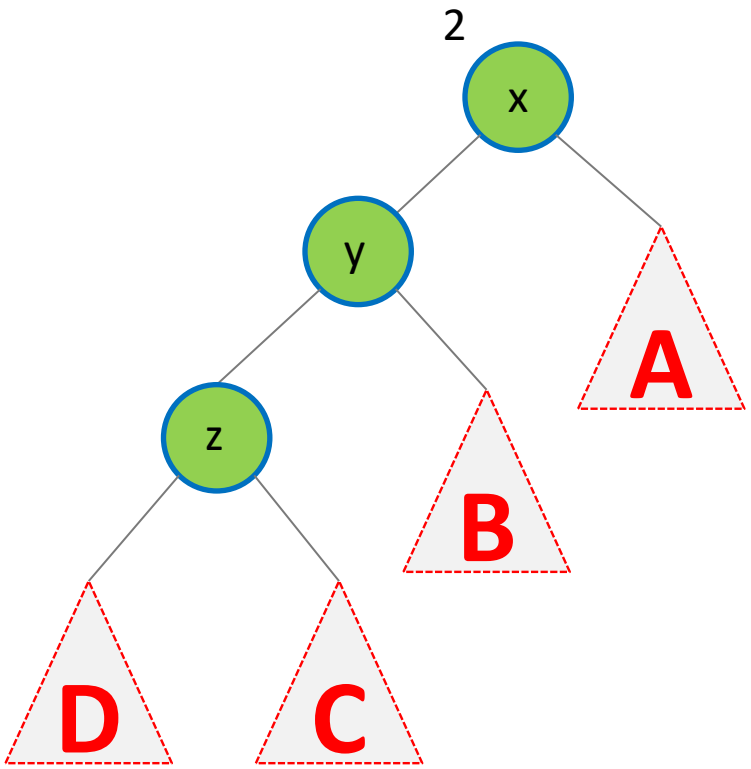

Rotate Y to be the parent of the subtree

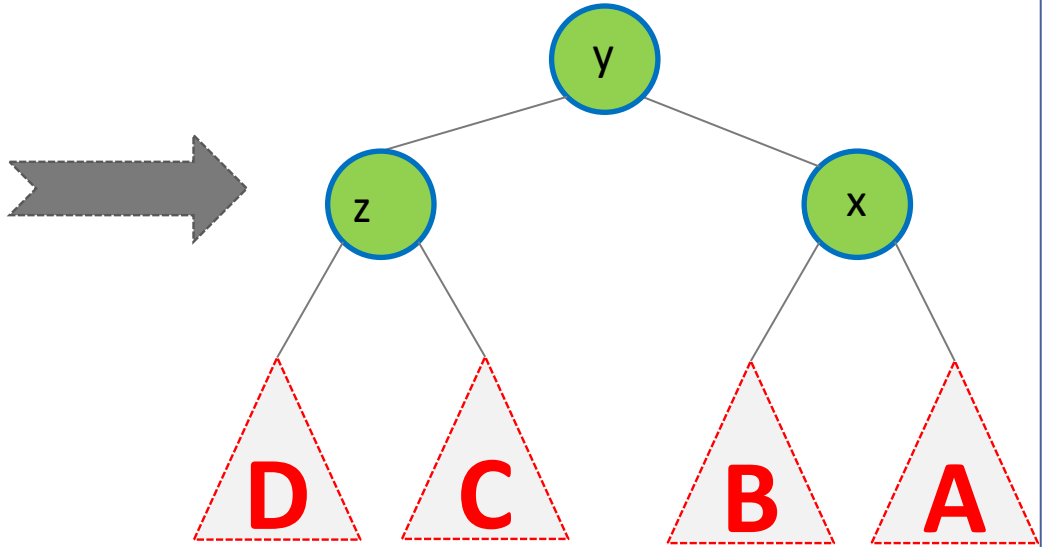



**What about B?**

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

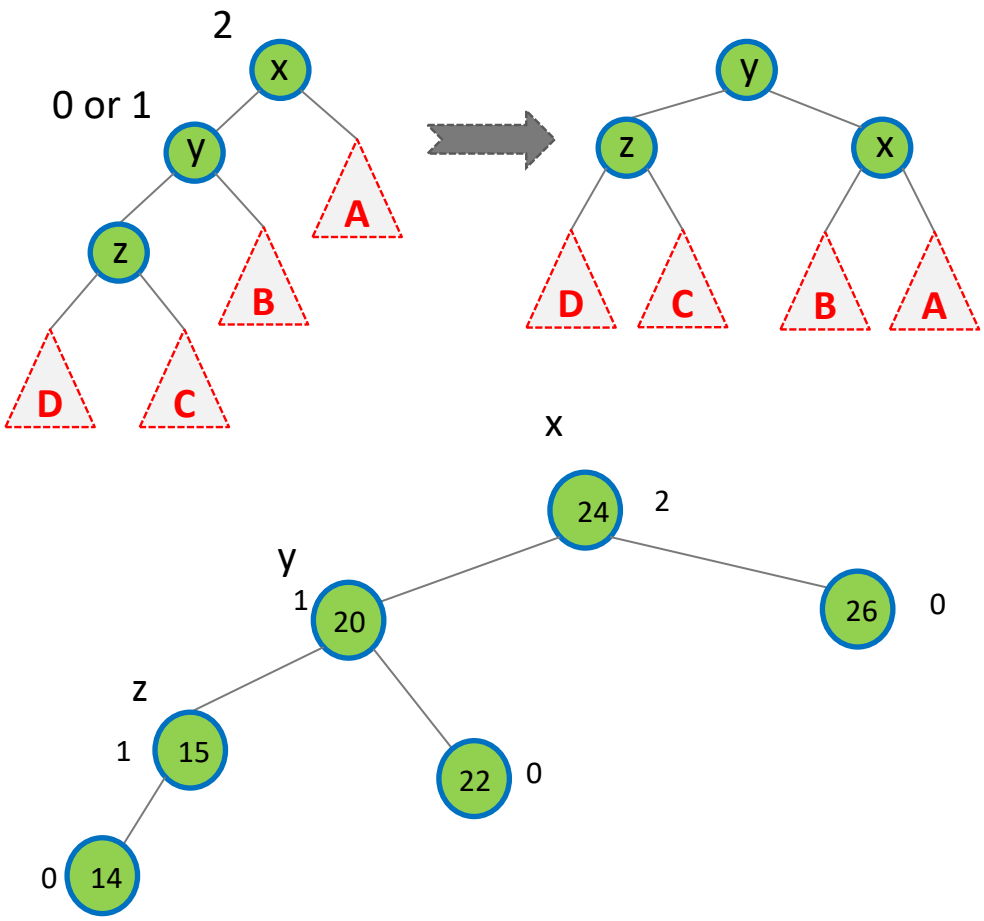
# Handling Left-left case



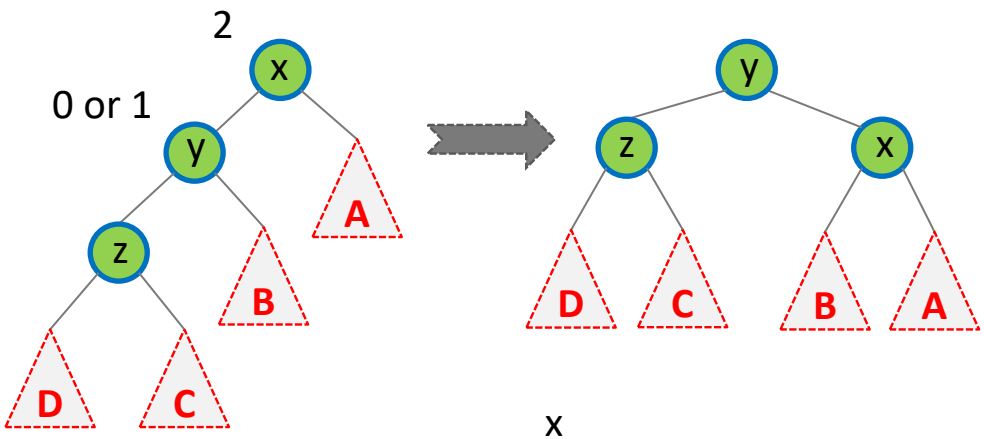



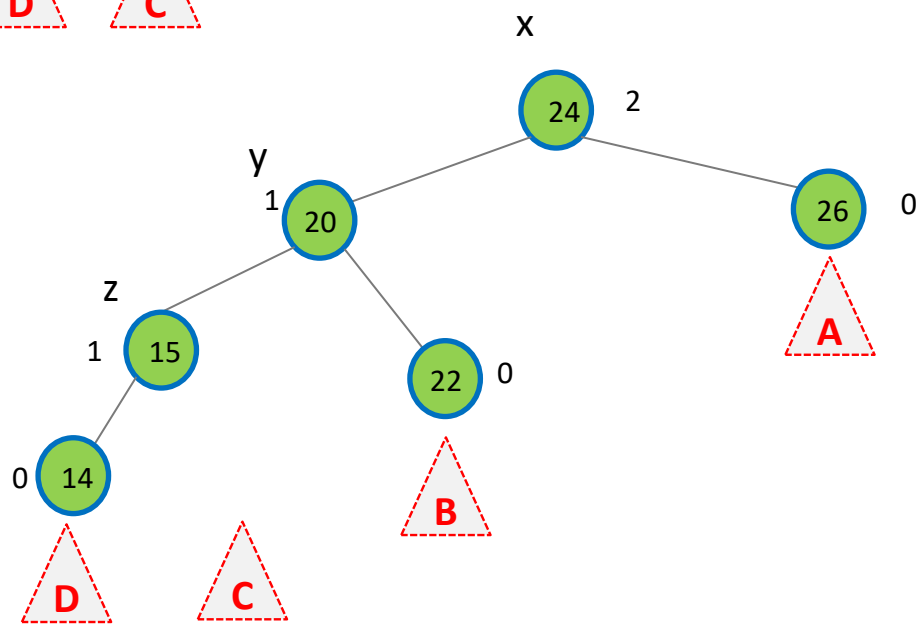

What about B? We have spare room at x
From “before” we know:
Values in B are > y
Values in B are < x
**B can be the left child of x!**

# Example: Left-left case

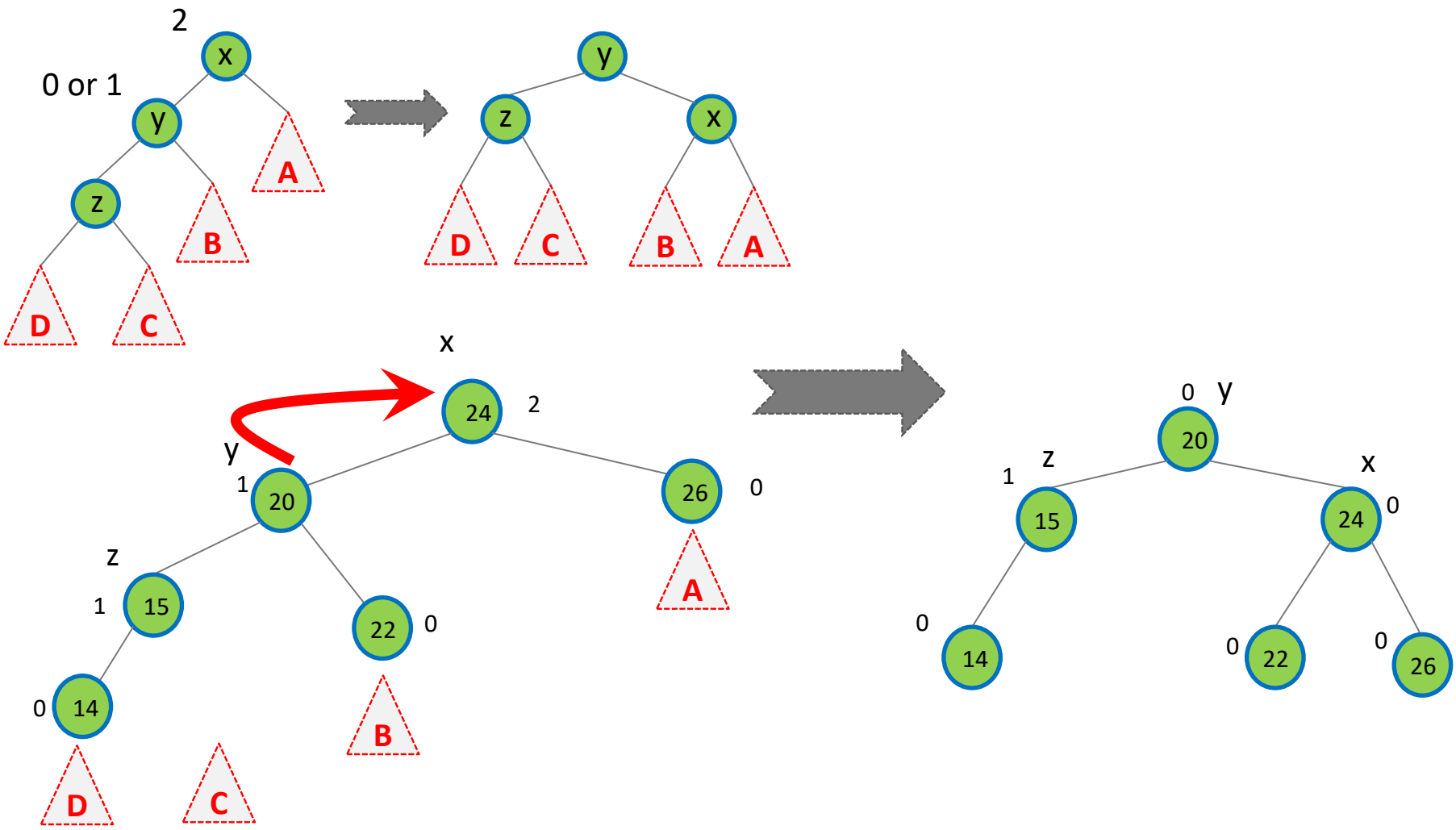


# Example: Left-left case

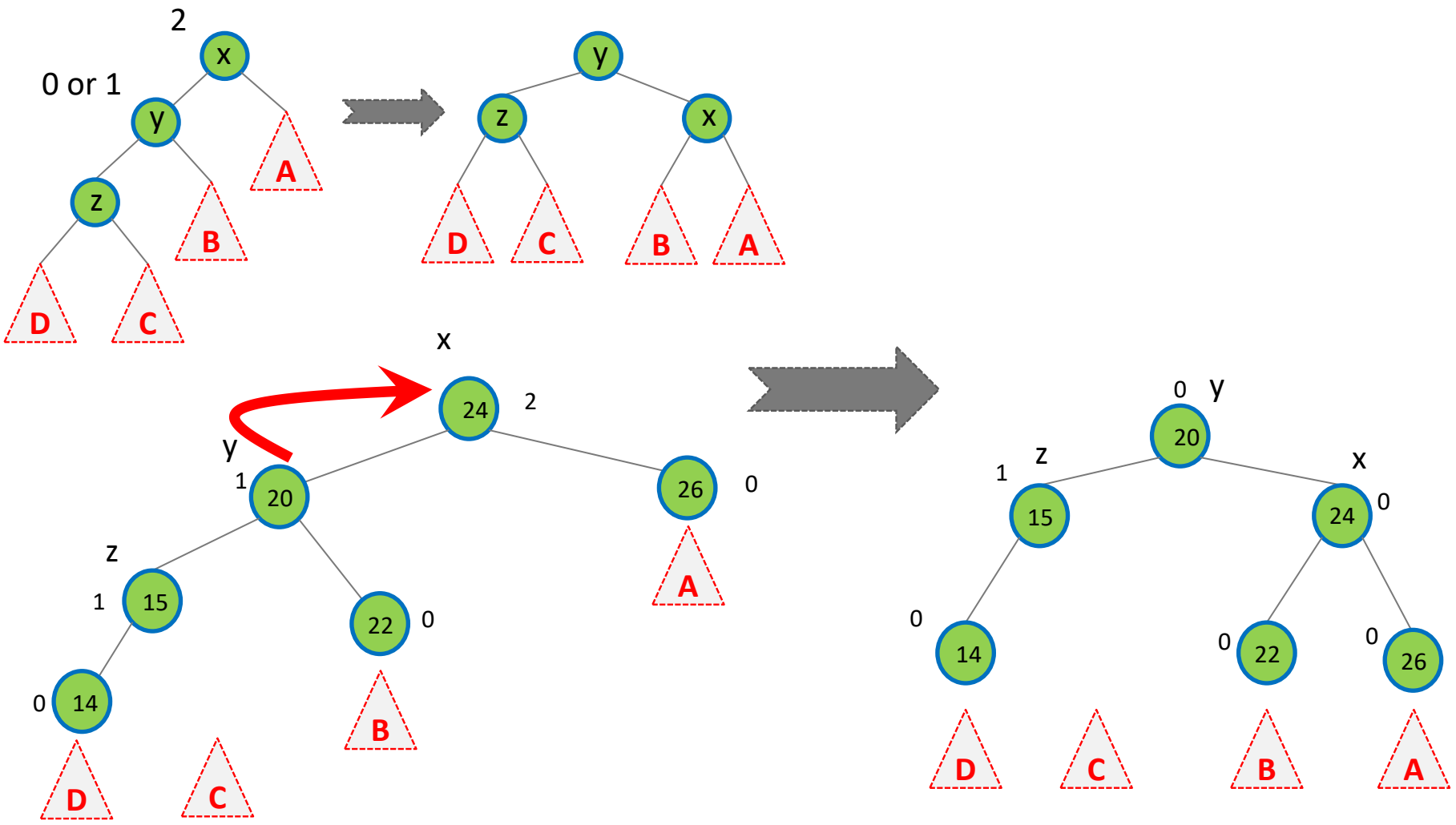


# Example: Left-left case

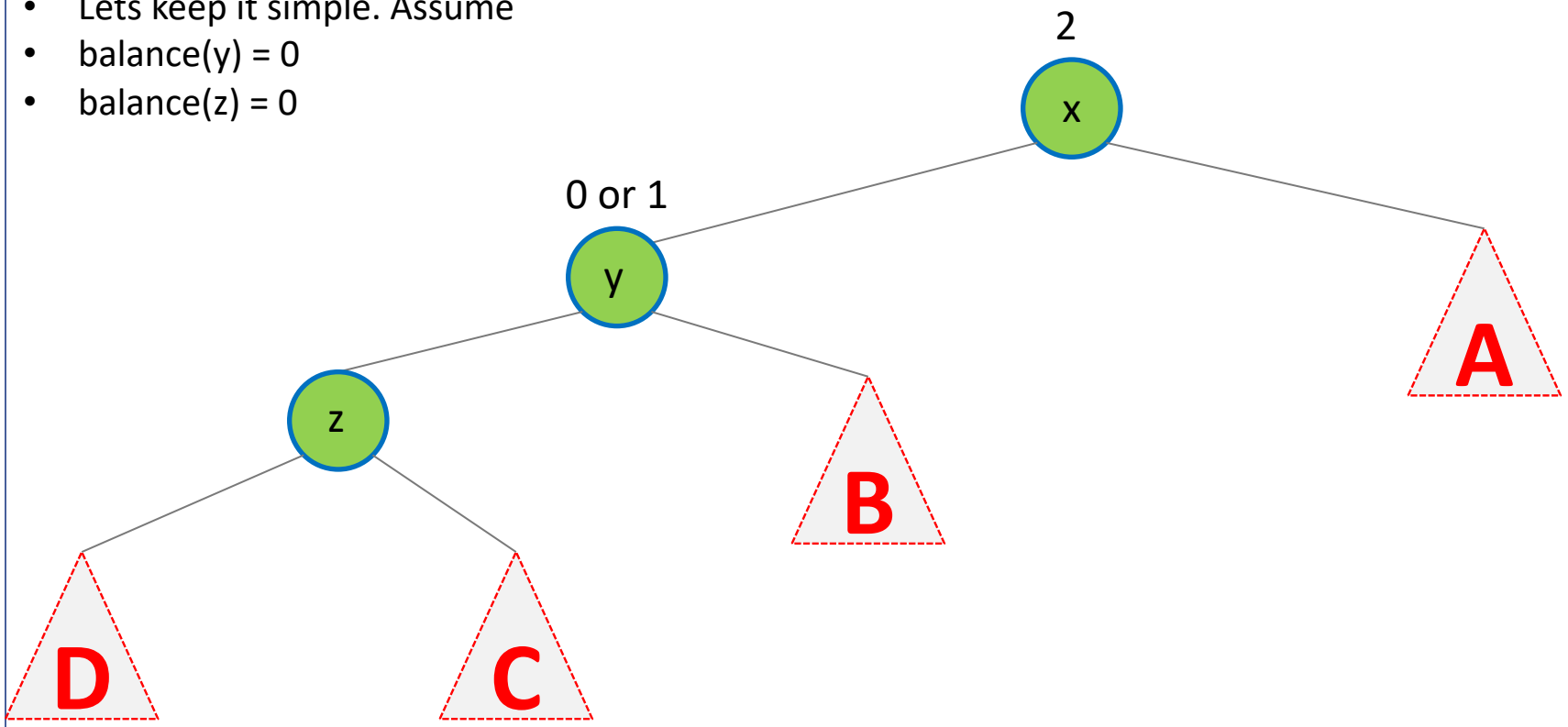


# Example: Left-left case



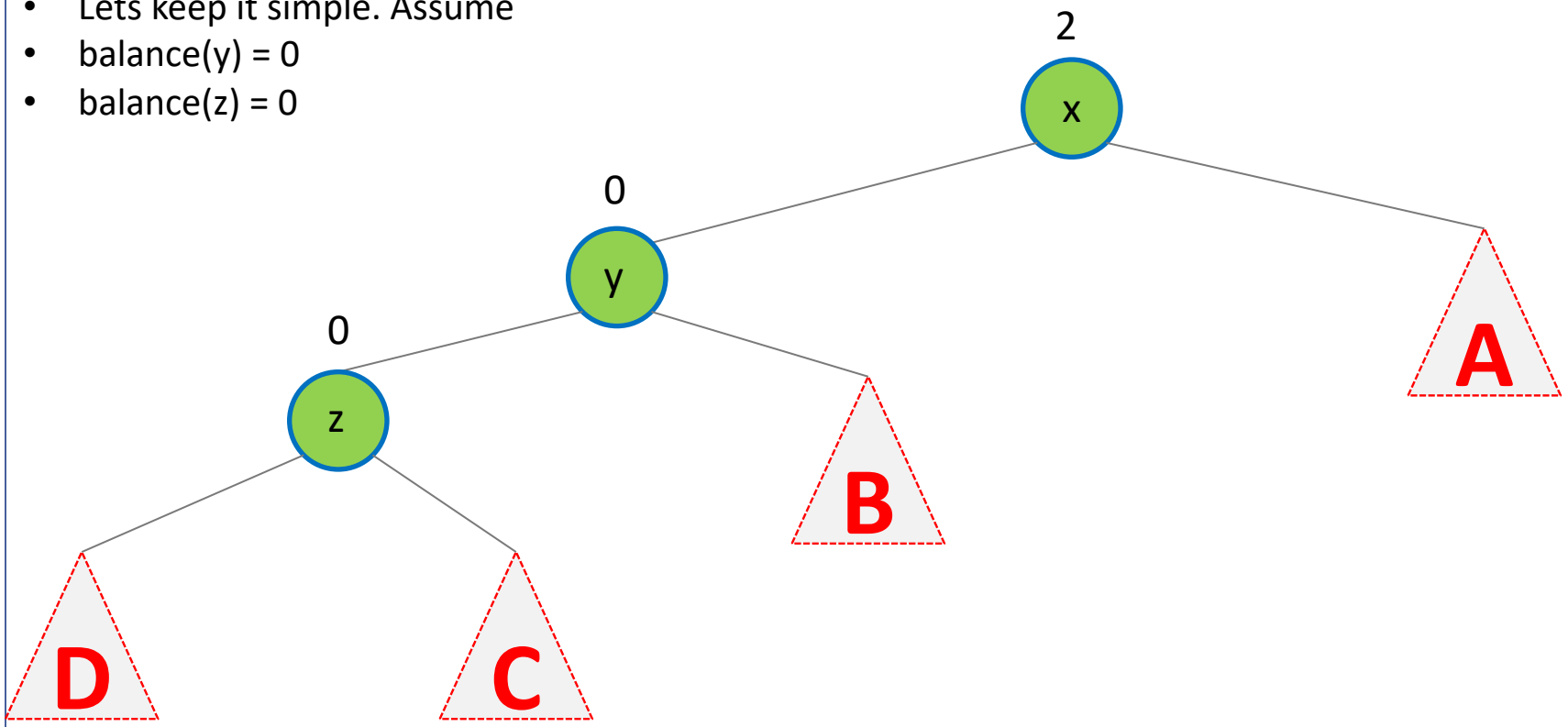
# Left-Left case: Does it work?

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



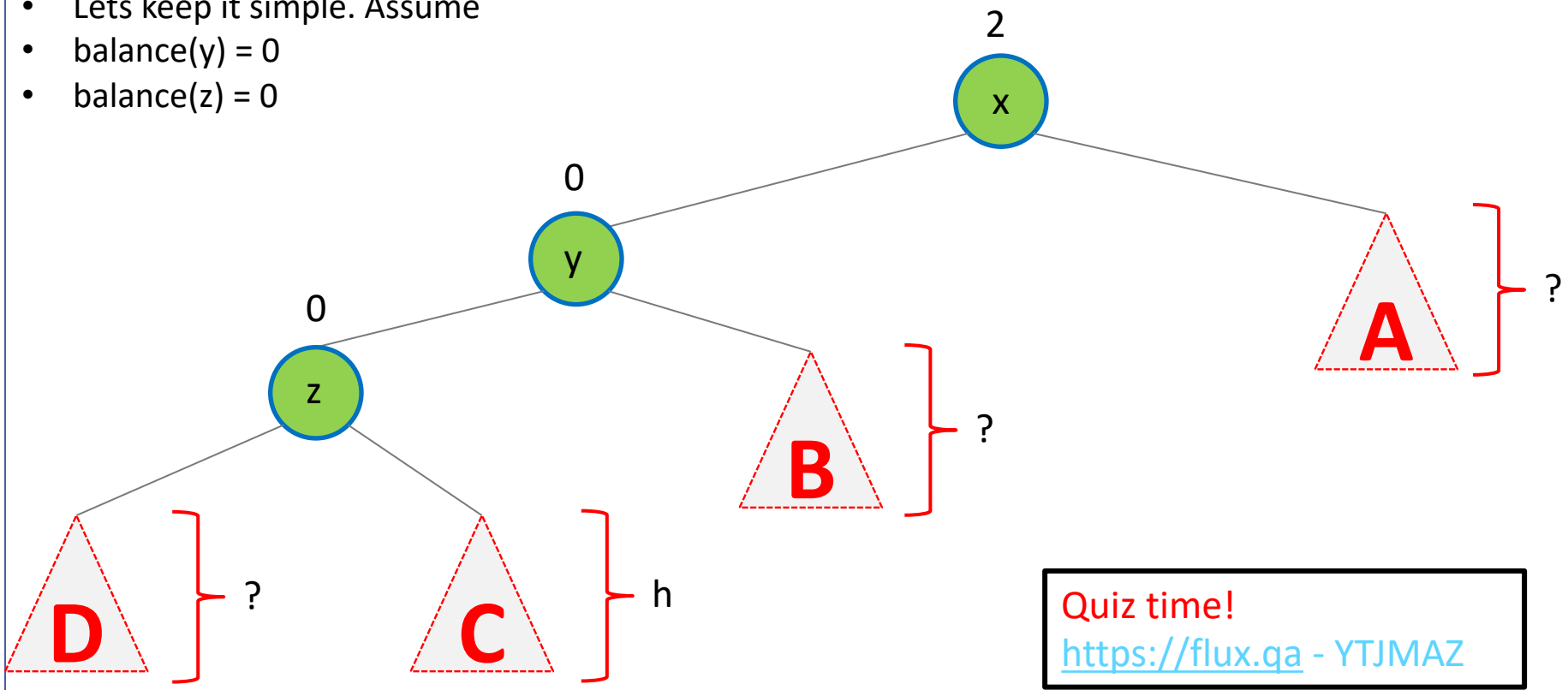
# Left-Left case: Does it work?

- 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?

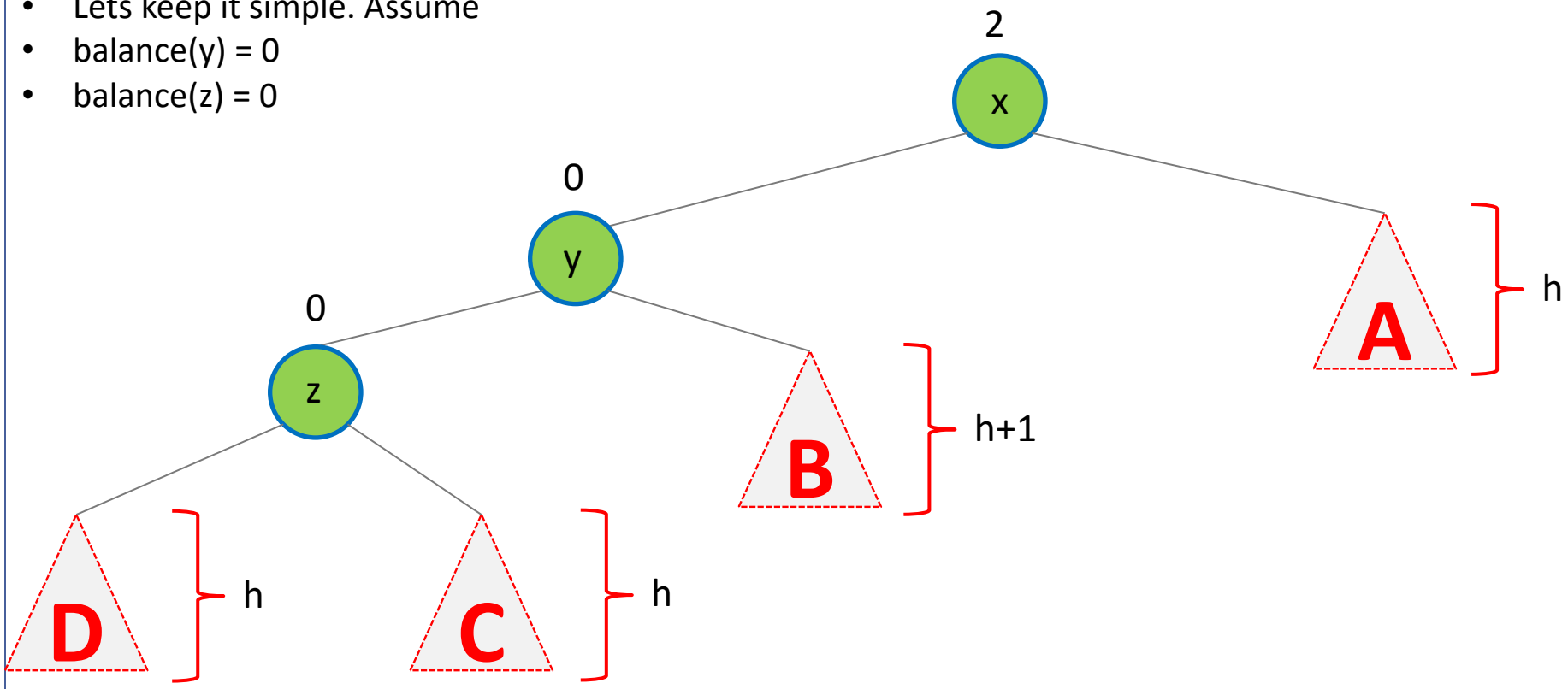
- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0




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

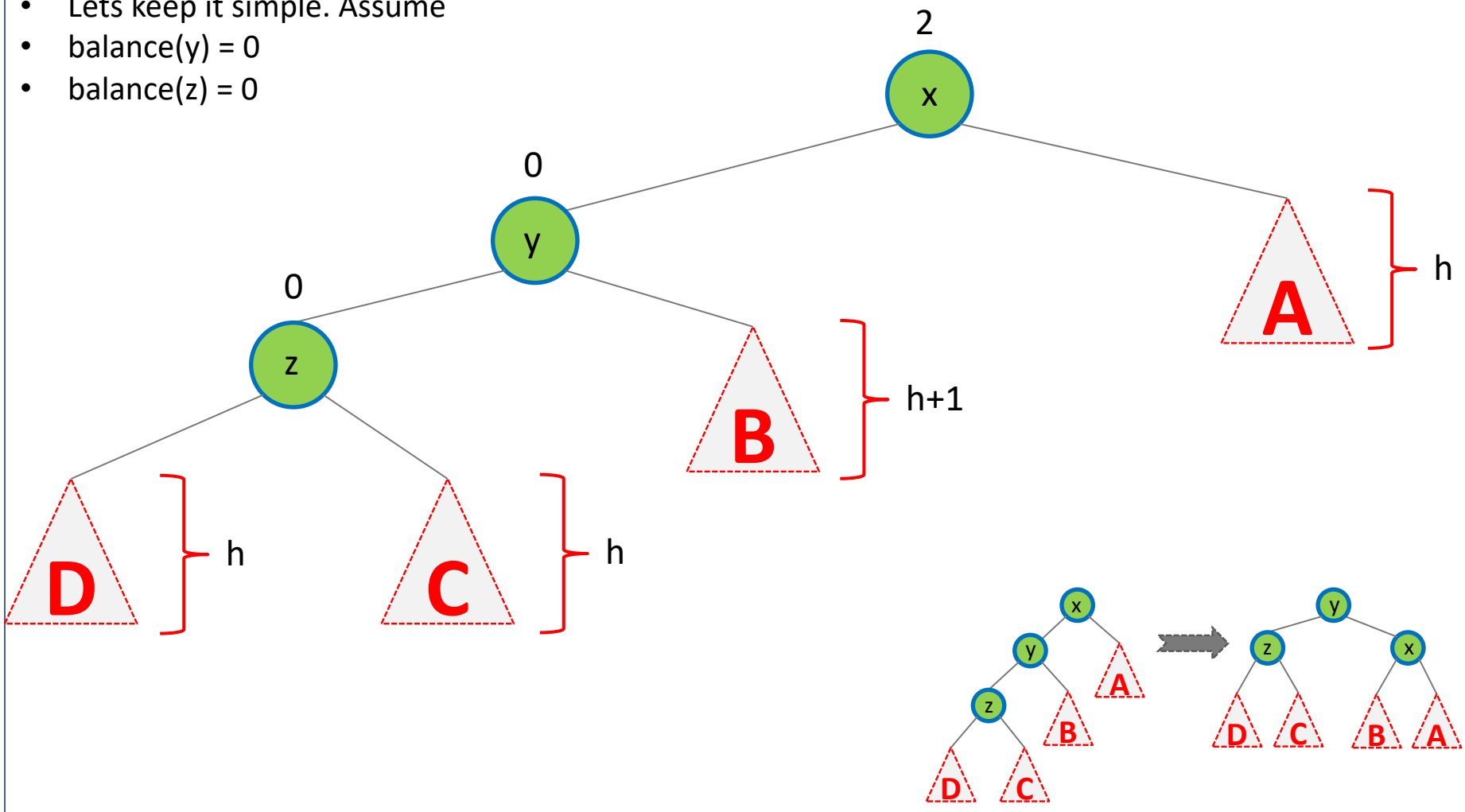
# Left-Left case: Does it work?

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



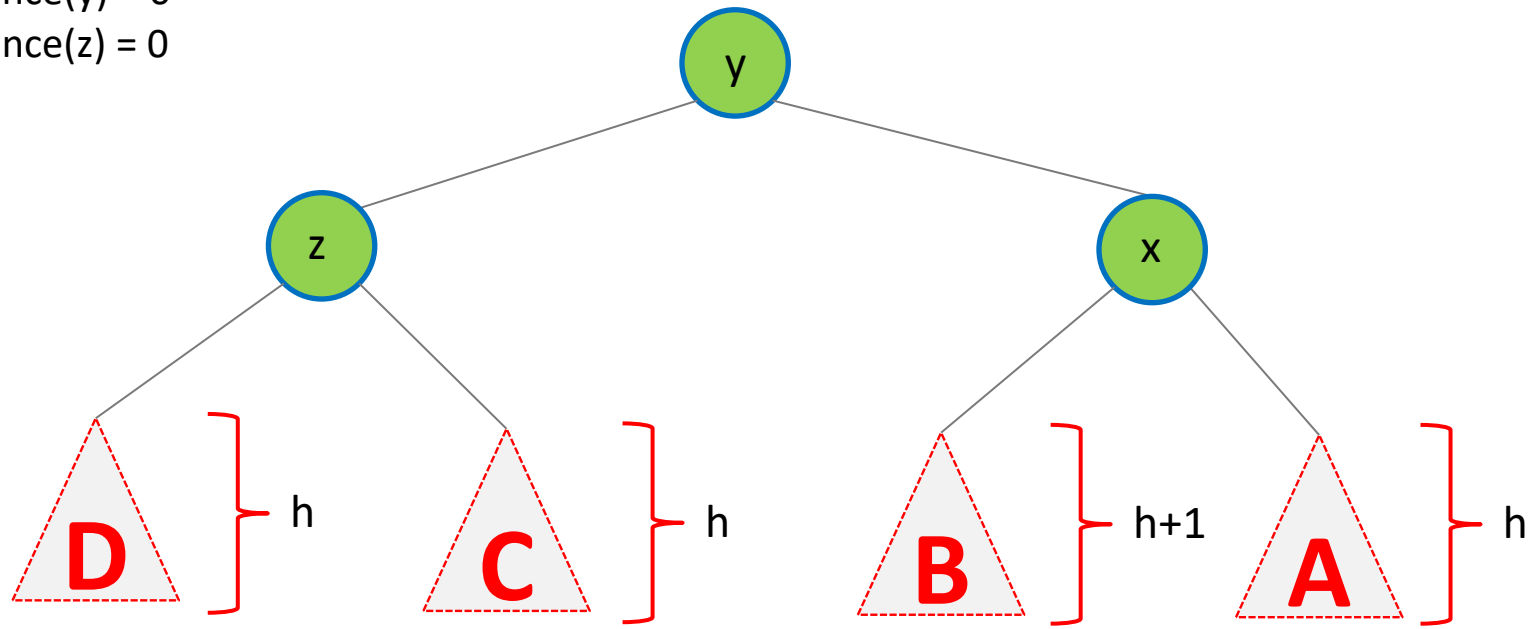
# Left-Left case: Does it work?

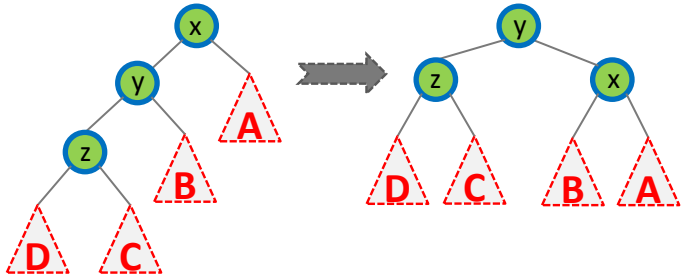
- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0



# Left-Left case: Does it work?

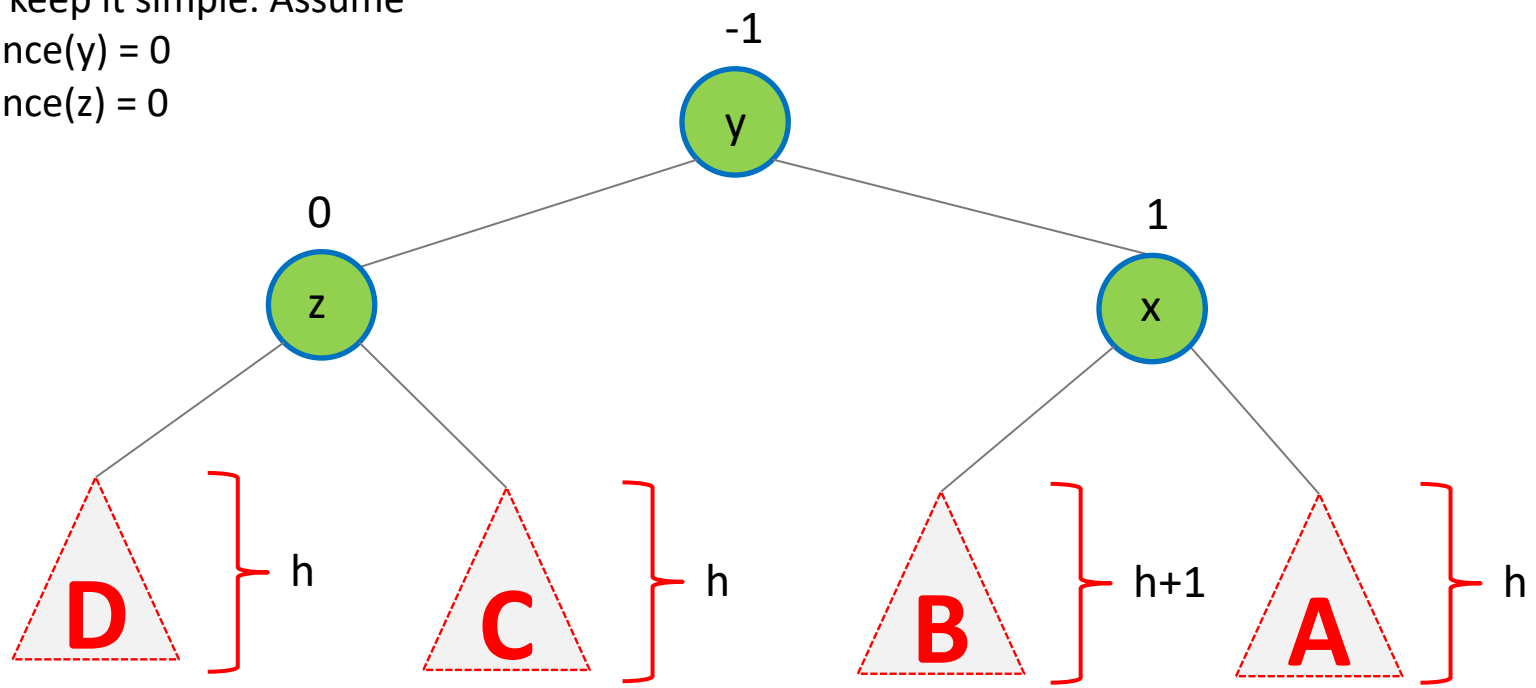
- Y could have balance factor 0 or 1
- Lets keep it simple. Assume
- balance(y) = 0
- balance(z) = 0

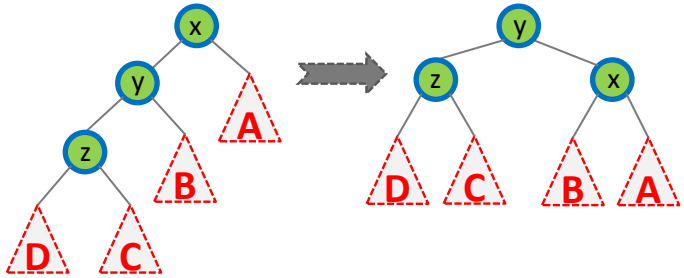


# Left-Left case: Does it work?

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



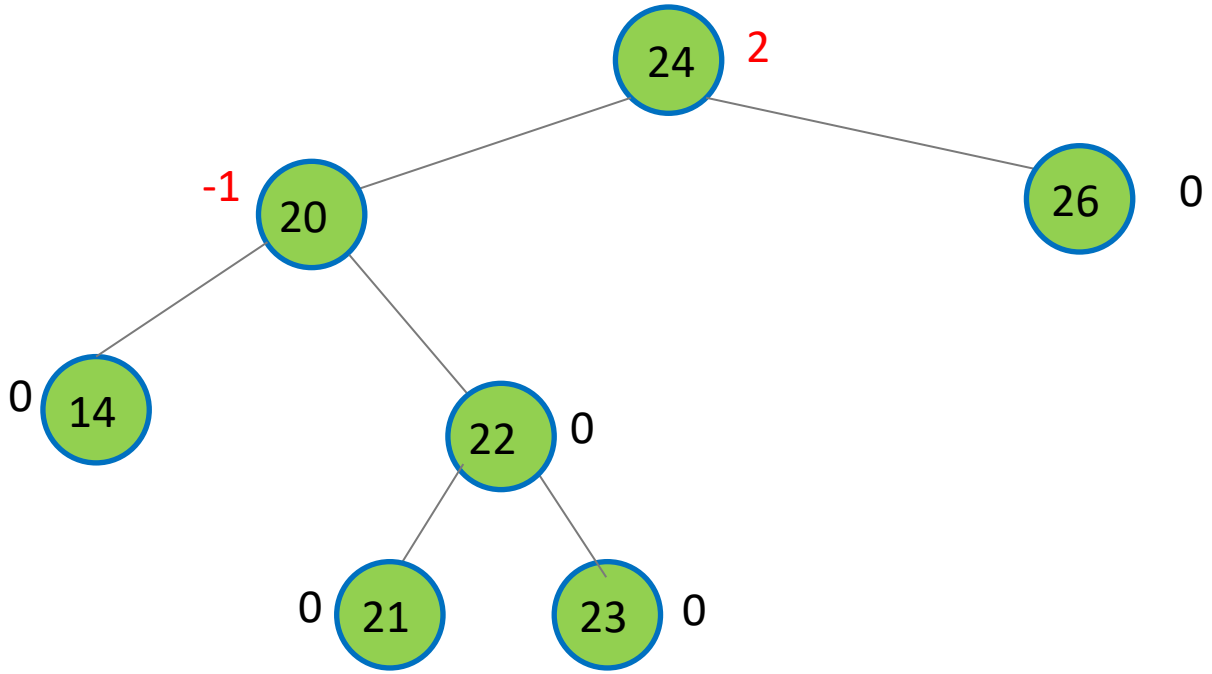
# Left-Left case: Does it work?

- 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
- The path to the deepest node turns left first, then right



# Left-Right Case

IMPORTANT

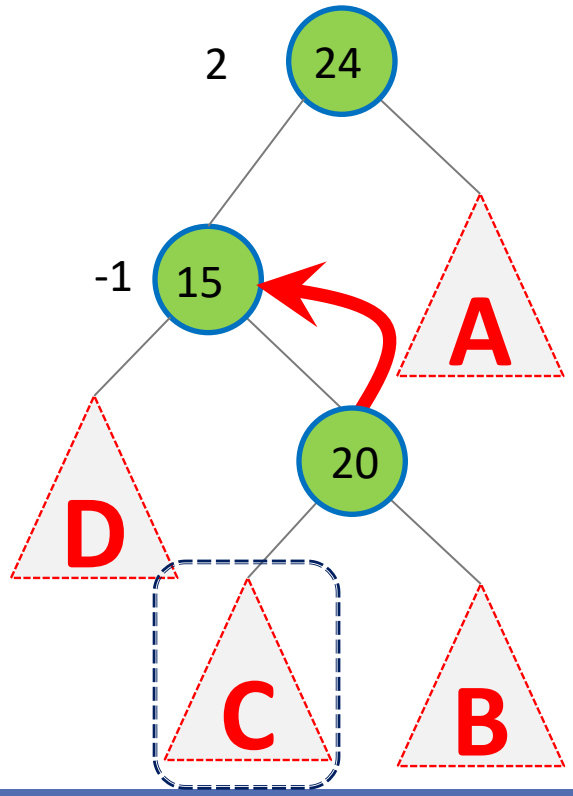
- Left-right cases are handled in a double-rotation (two steps)
- Execute these as an atomic operation
  DO NOT stop in between
- After step 1, you may have created other problems with balance factors, but keep going and do Step 2 before checking the state of the AVL tree

- The idea is to use one rotation to reduce to the left-left case (we know how to handle that!)

# Handling Left-right case

**Left Right Case**

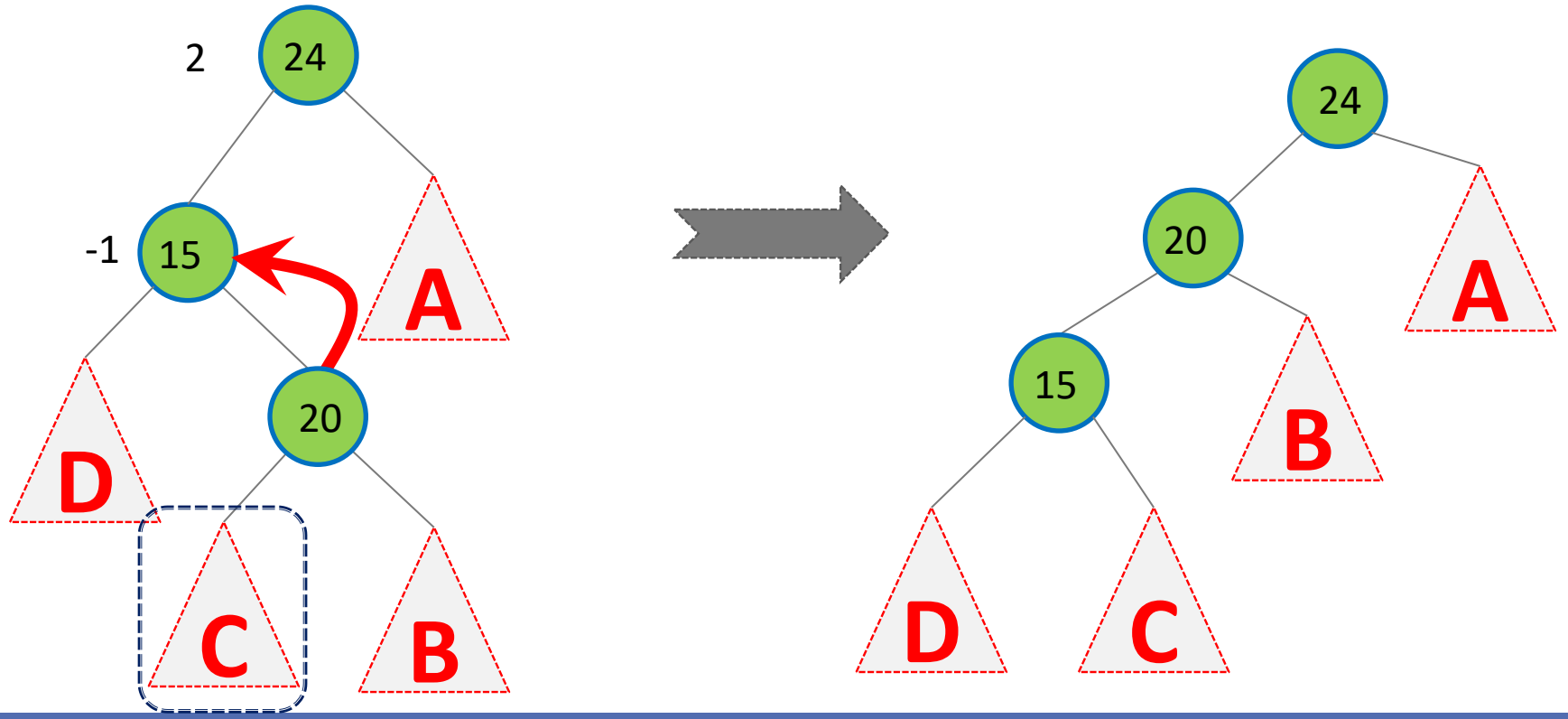
1. Convert Left Right case to Left-Left case by rotating 20.



# Handling Left-right case

**Left Right Case**

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

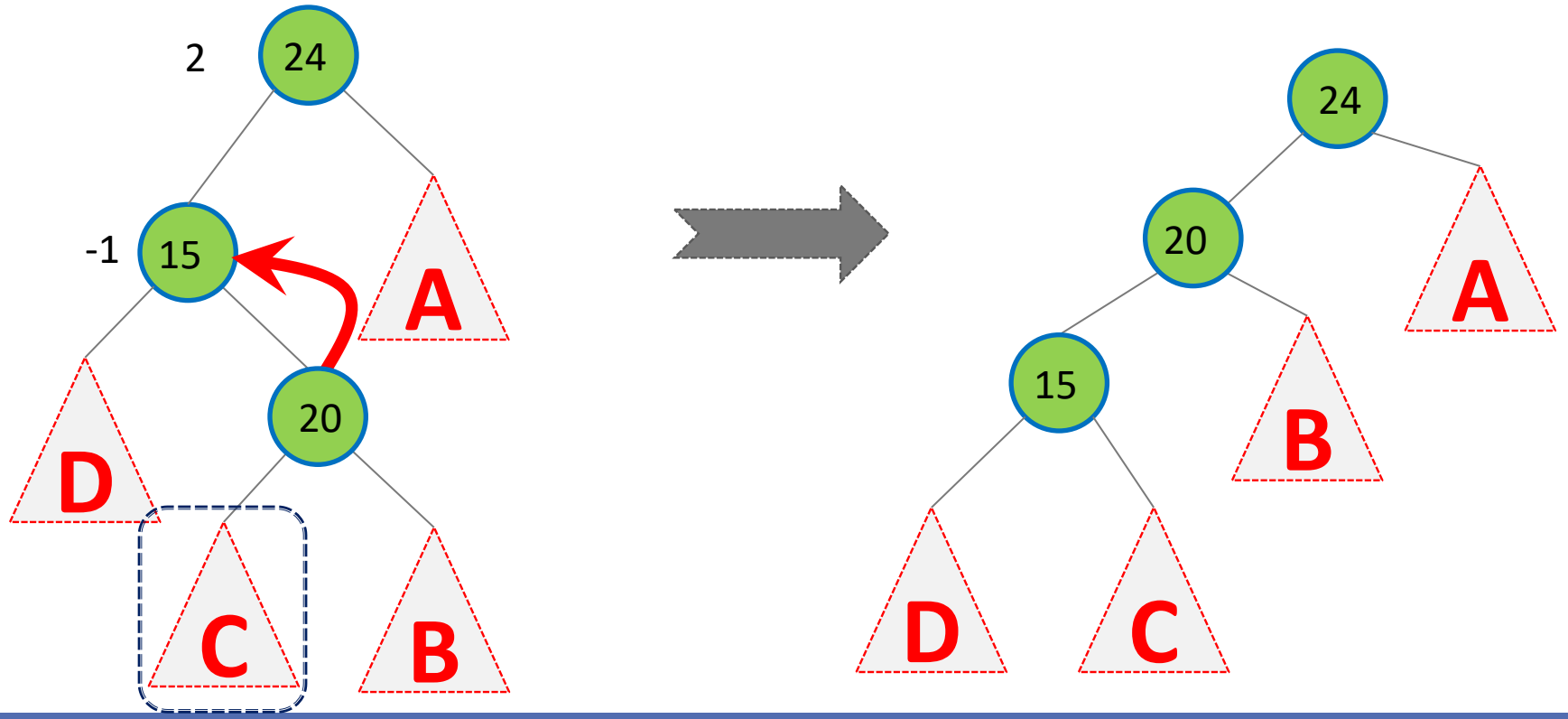


# Handling Left-right case

**Left Right Case**

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

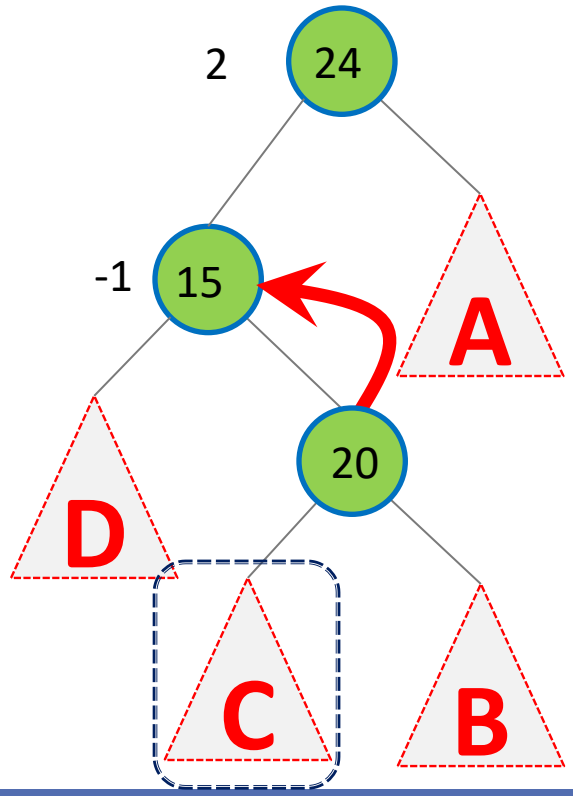
Now we have a left-left case!



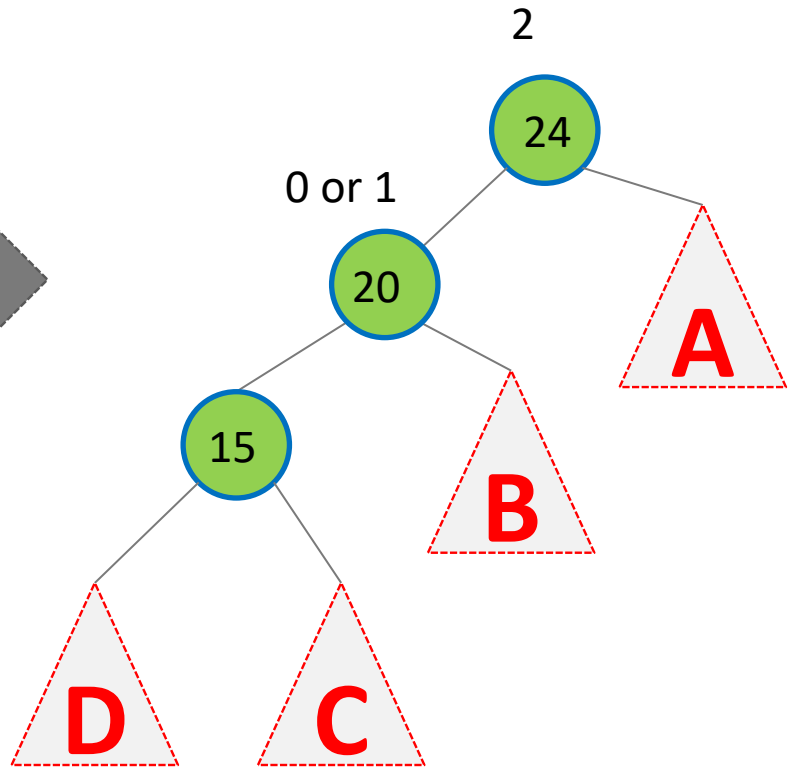
# Handling Left-right case

**Left Right Case**

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



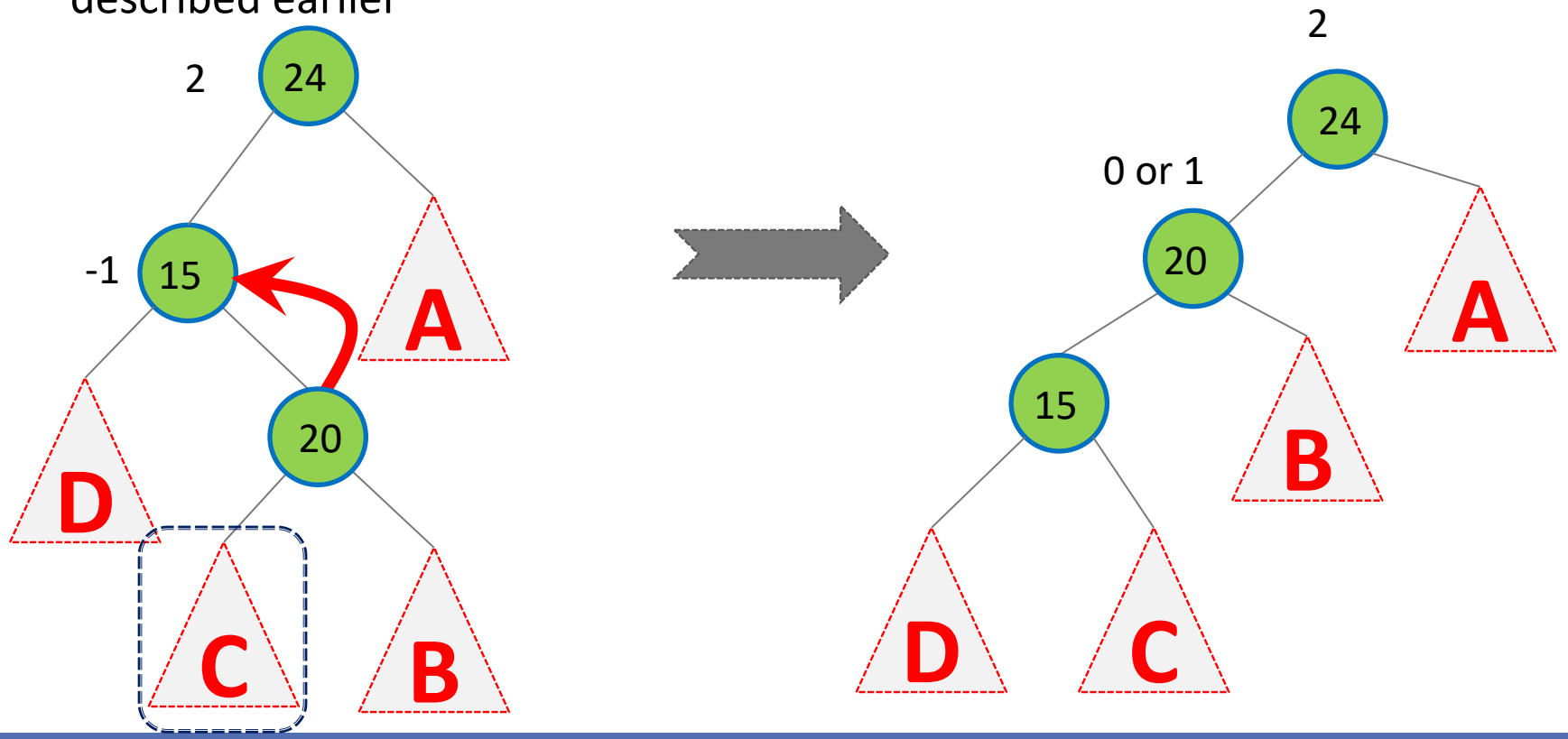

Now we have a left-left case!



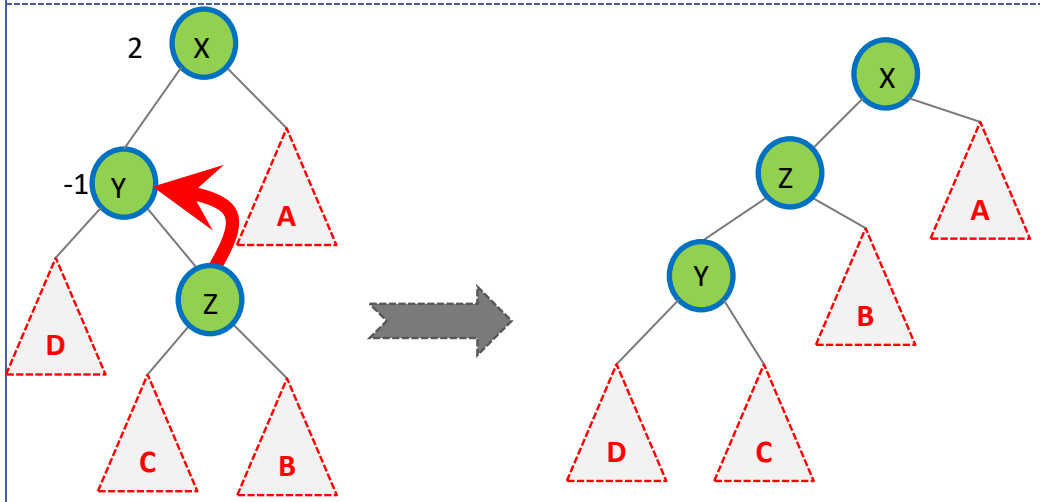
# Handling Left-right case

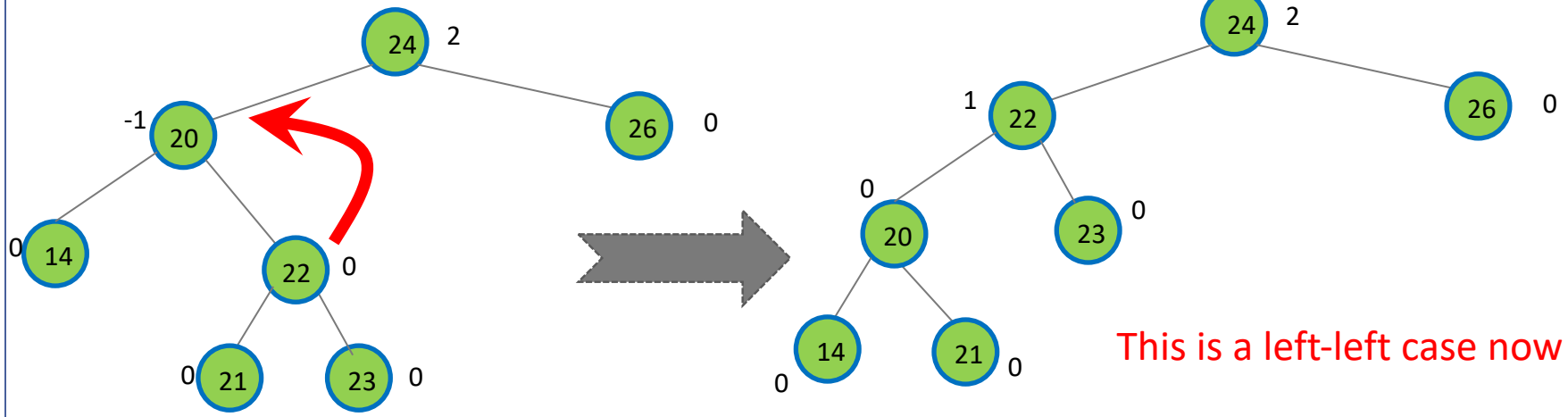
**Left Right Case**

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



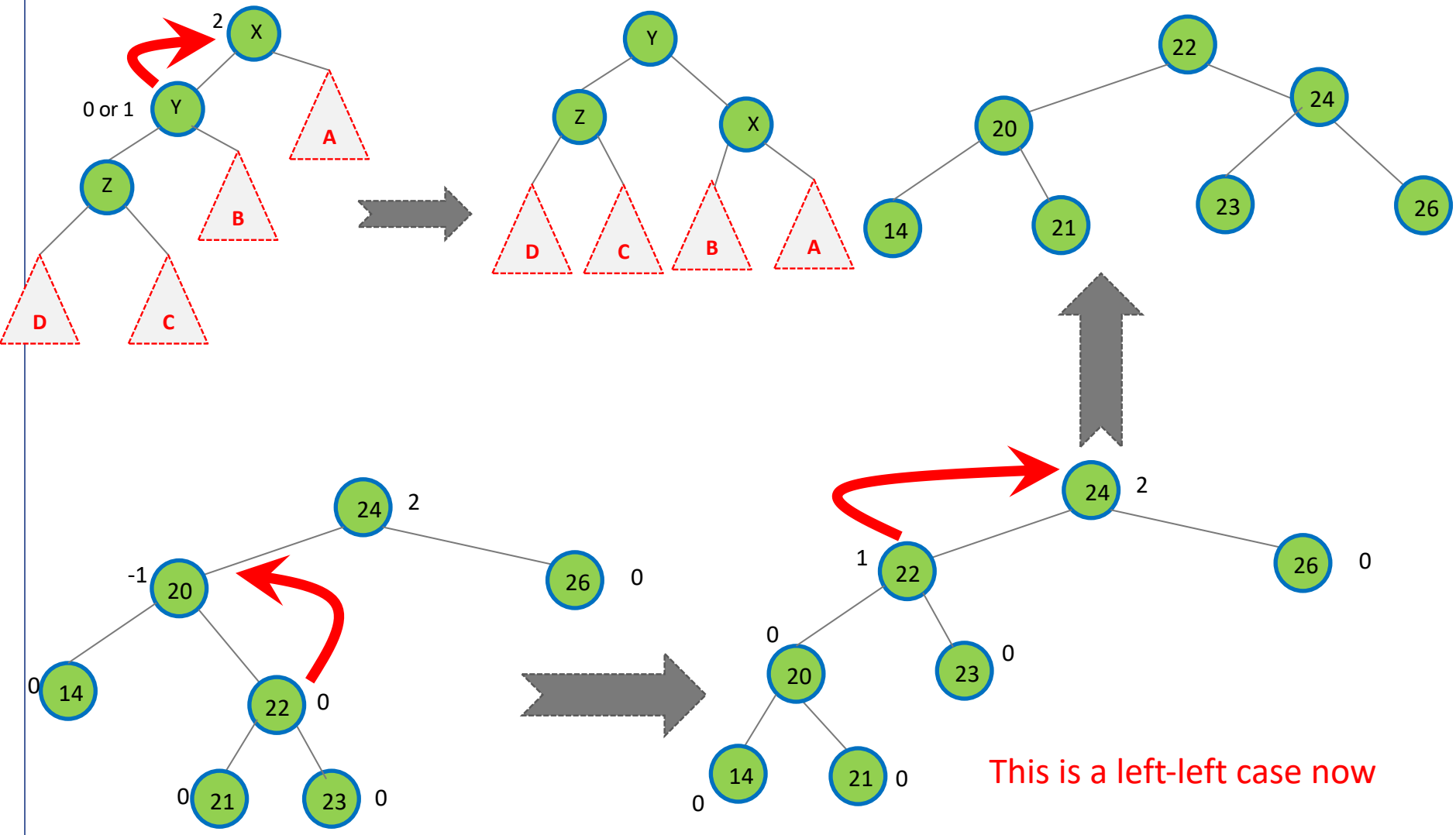
# Example: Left-right case – Step 1




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

# Generalised Trinode Restructuring

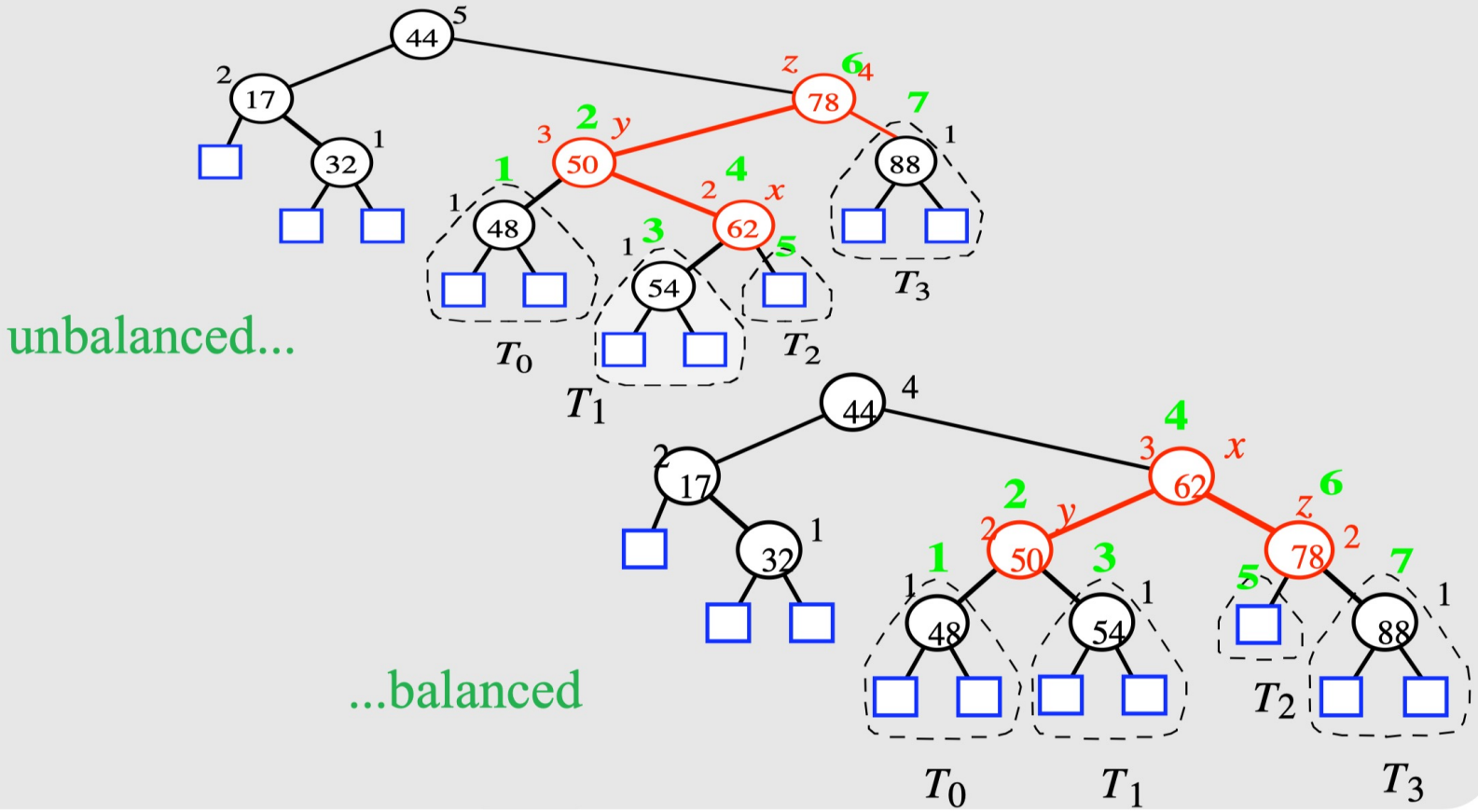
Note that all the operations above can be summarised as one single case:

1. let z be the first imbalanced node,
2. y its higher child and
3. x the higher child of y
4. let a, b, c be an inorder enumeration of x, y, z
5. let T0, …, T3 be the left-to-right listing of the subtrees rooted at x, y, z

Alg trinode-restructure

1. replace subtree rooted at z with subtree rooted at b
2. let a be the left child of b, and let T0 (T1) be the left (right) child of a
3. let c be the right child of b and T2 (T3) be the left (right) child of c

# Generalised Trinode Restructuring



# Outline

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

# Complexity of Balancing the AVL Tree

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

- O(log N) in the worst case because the tree is balanced
  (cf supplementary questions Tute 6)

Insertion/Deletion in AVL Tree

- Insert/Delete the element in the same way as in BST – O(log N)
- Balance the tree if required

Worst-case insertion/deletion time complexity: ??

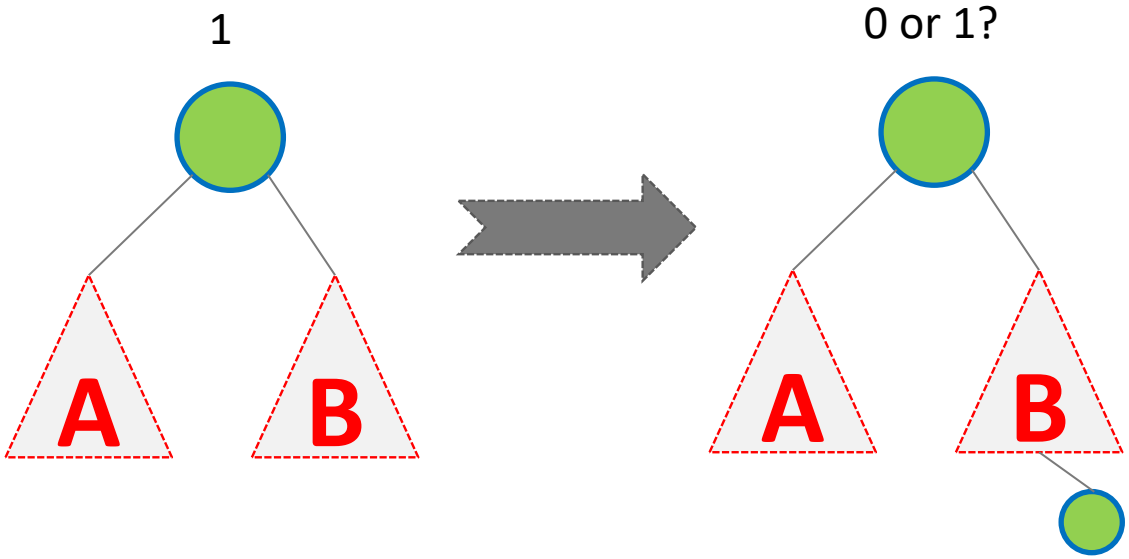
# Complexity of Balancing the AVL Tree

The tree is balanced bottom-up starting from the lowest unbalanced node

- After each (double)rotation:
  - We have restored the balance at the node where we restructure,
  - We may not have restored (or upset) the balance at higher nodes (witness trinode restructuring example “insert 54”)
- After a insert / delete we may thus have to check all the nodes on the path back to the root
- We need to balance at most O(log N) nodes in the worst-case
- Balancing at each node takes constant time (relinking)
- So total cost of balancing after insertion/deletion is O(log N) in worst-case

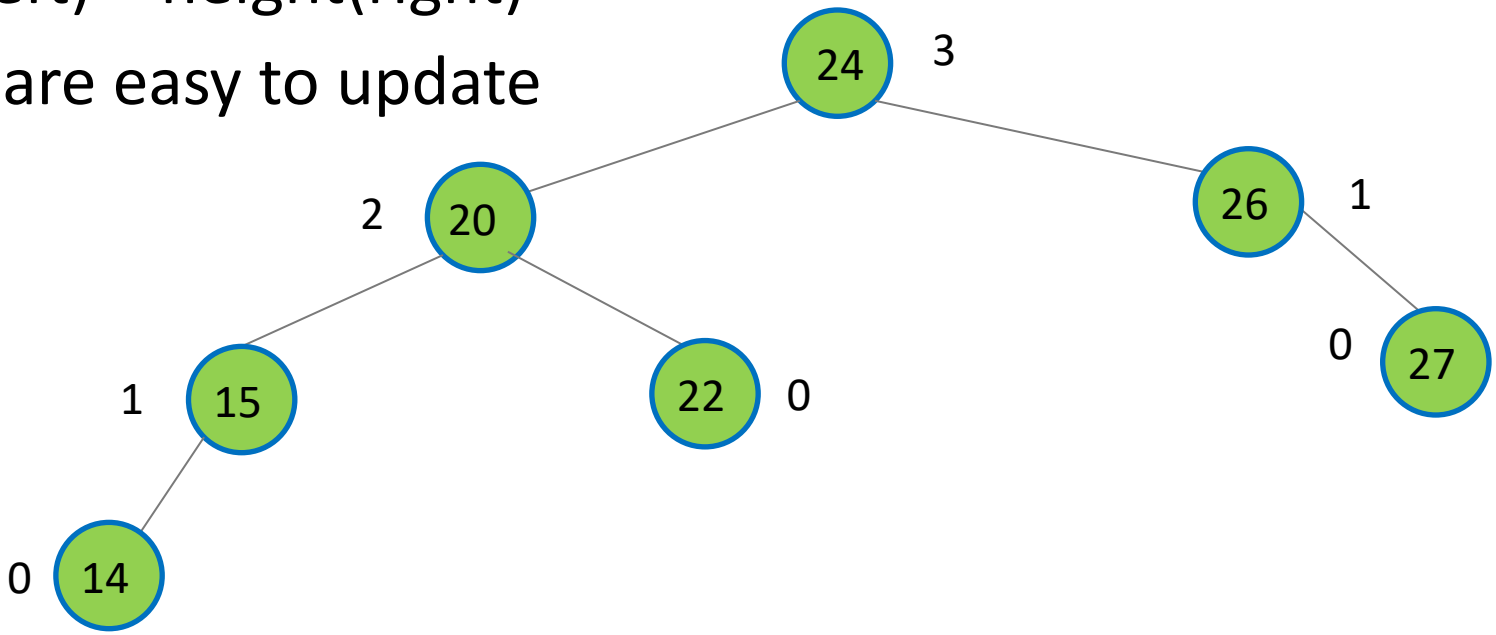
# Computing balance factors

- How do we update balance factors quickly?



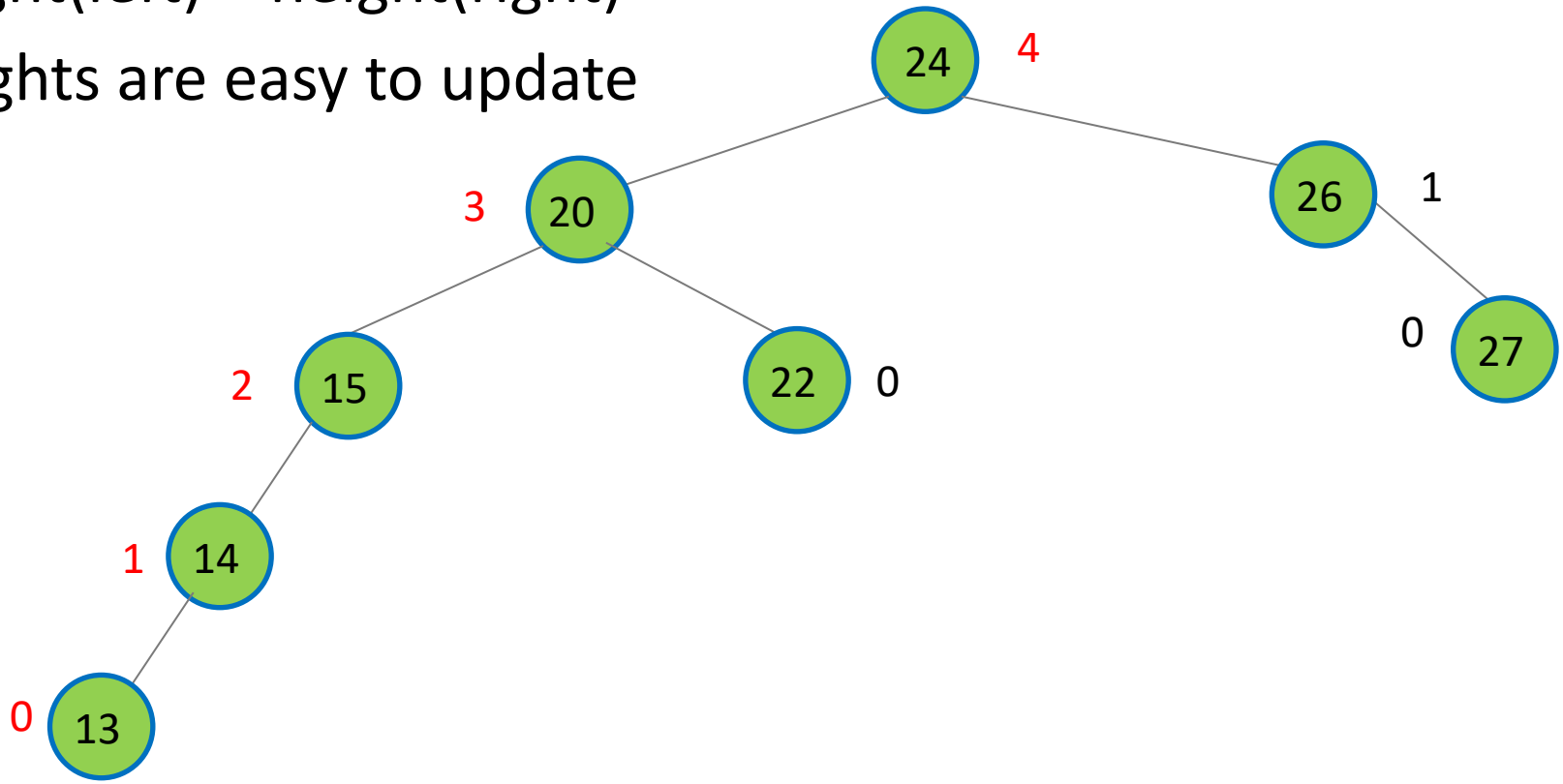
# Computing balance factors

- Instead, store heights at each node
- Balance can then be computed in O(1) as height(left) – height(right)
- Heights are easy to update



# Computing balance factors

- Instead, store heights at each node
- Balance can then be computed in O(1) as height(left) – height(right)
- Heights are easy to update



# The Zoo of Self-balancing Trees

- We cannot fully rebalance a binary tree after every operation. This is too costly.
- Thus, we cannot keep a binary tree perfectly balanced at all times.
- We can relax the requirement in three different ways still reach O(log n) access times.
  - *almost balanced at all times* (AVL Trees)
  - *almost balanced most of the time* (Splay Trees)
  - perfectly balanced at all times but allow it to be it *non-binary* (2-4-Tree, Red-Black Tree *(sort of)* ...)

# Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables
   A. Introduction
   B. Chaining
   C. Probing
      I. Linear Probing
      II. Improved hashing [Quadratic Probing, Double and Cuckoo Hashing]
5. Randomised Lookup

# Direct Array 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]

# Direct Array Addressing

What if our keys are not numbers?

Hashing: Compute a numerical index from the key.

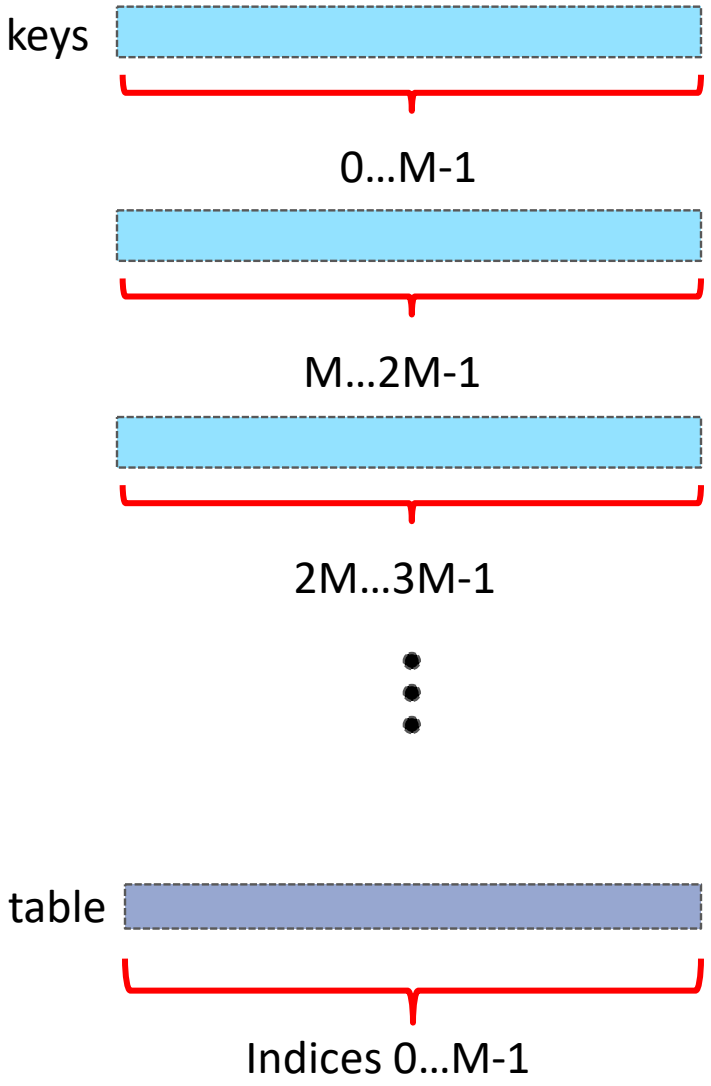
Take strings as an example:

- Convert letters to (8-bit) ascii values
- Take this as a number in base 256?
- Add them all together?
- Multiply them all together?
- Do something more complicated?

This can lead to **huge and very sparse** arrays!

- we may not even know the largest index

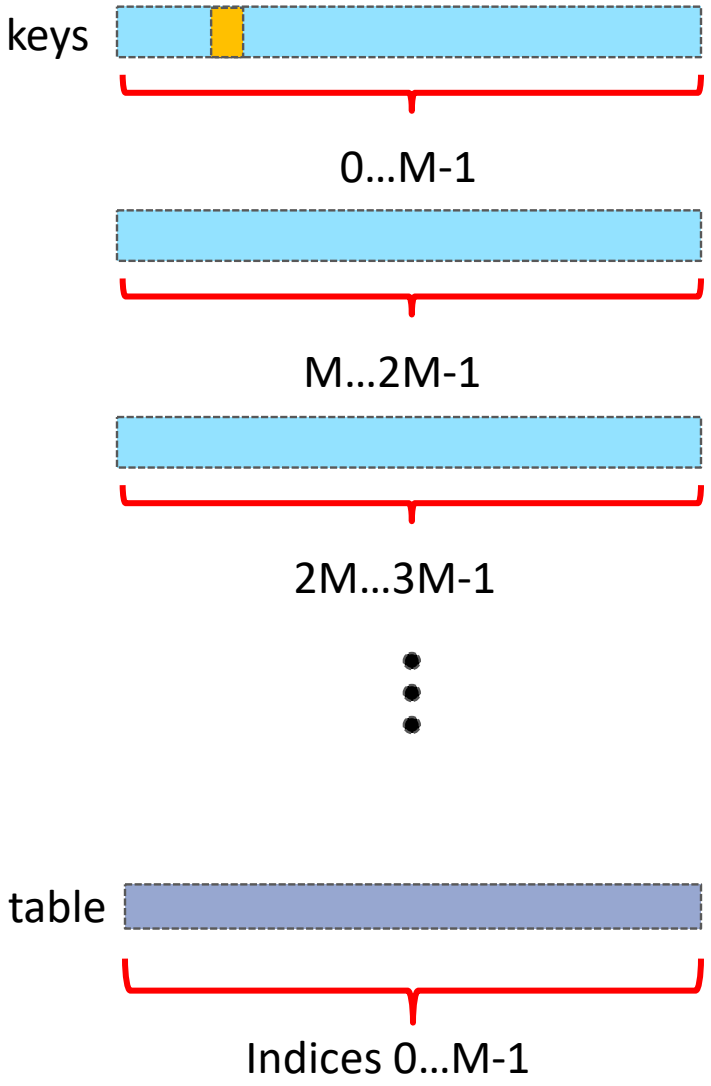
# Restricting to a know range




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

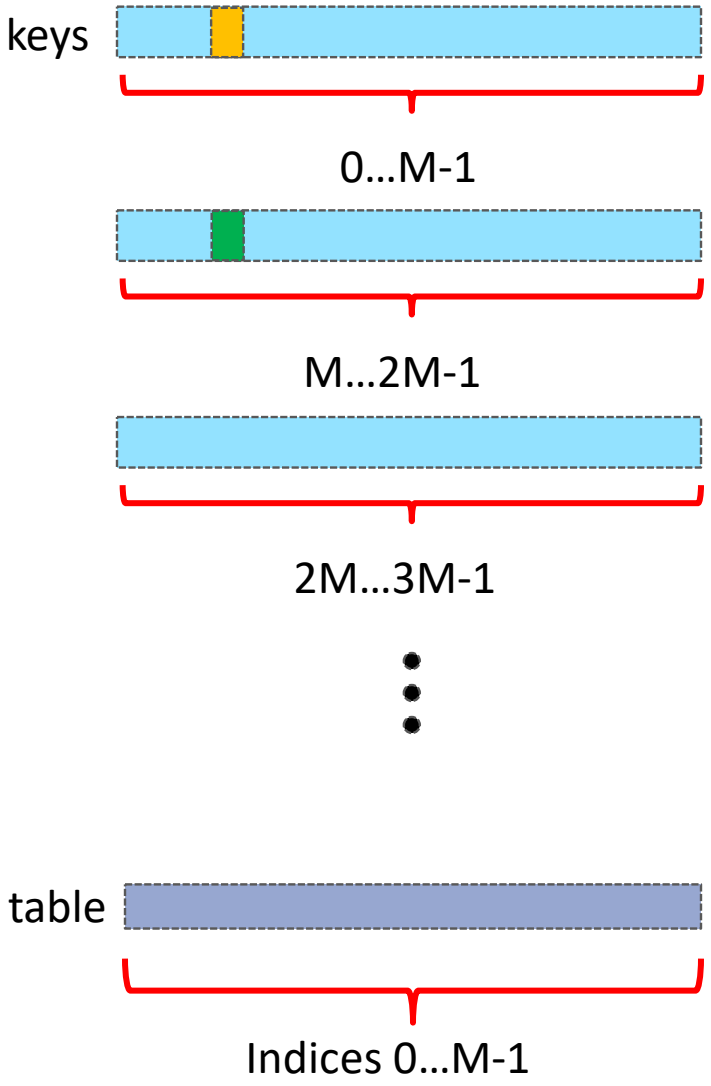
- One way to do this in practice is to use modulus (%)
- **Note that just taking keys % m is a bad hash function!**
- **a%m** = the remainder when **a** is divided by **m**
- Always gives a value in the range [0..m-1]

# Collisions



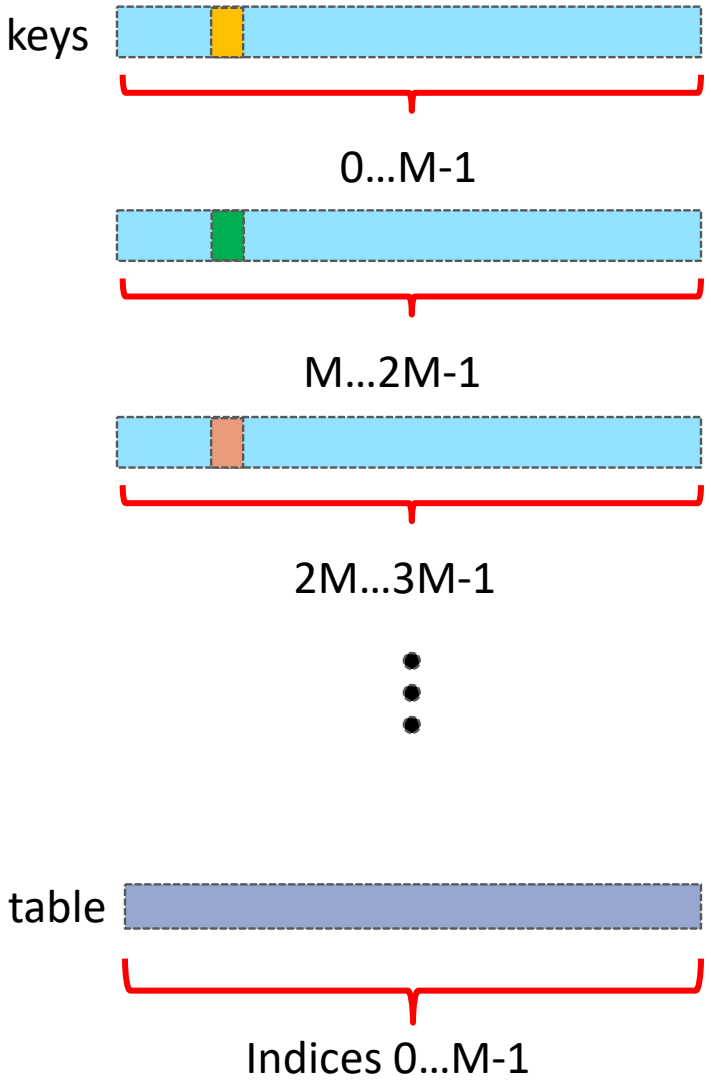

- An unavoidable consequence of having more keys than indices in our table

# Collisions



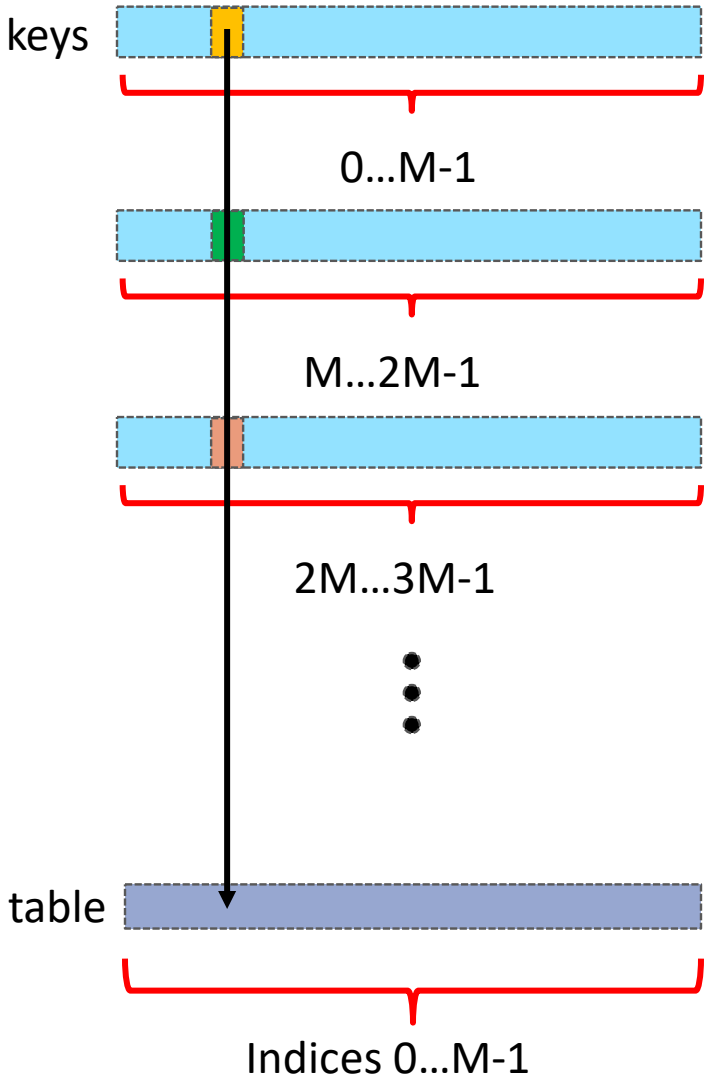

- An unavoidable consequence of having more keys than indices in our table

# Collisions




- An unavoidable consequence of having more keys than indices in our table

# Collisions




- An unavoidable 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 - YTJMAZ

- 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
5. Randomised Lookup

# Chaining

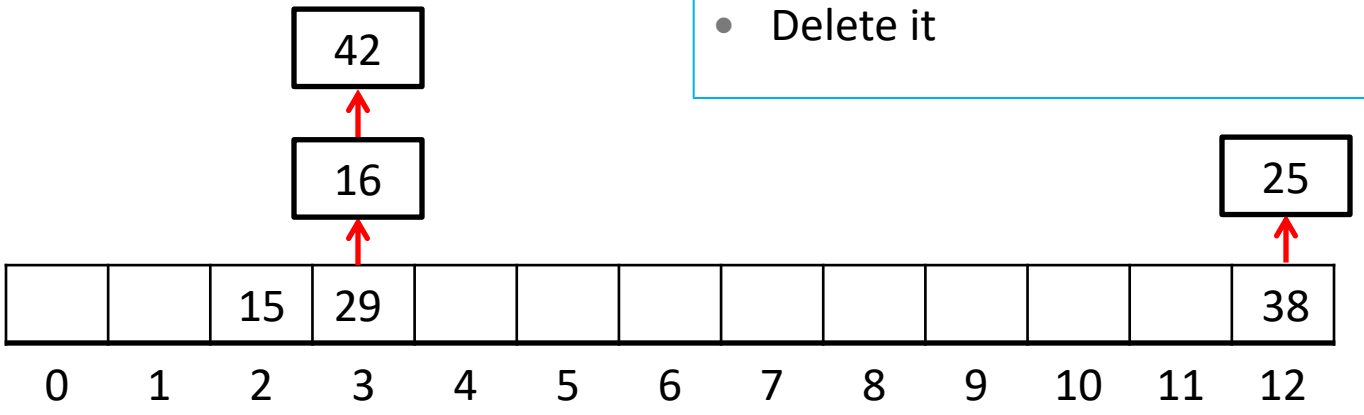
- Allow unlimited storage at every index (linked list)
- If there are already some elements at hash index
  - Add the new element in a list at array[index]
- Example: Suppose the hash table size M is 13 and hash function is key % 13.
  - Insert 29 → index = 29%13 = 3
  - Insert 38 → index = 38%13 = 12
  - 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



# Average Case - 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 (utilisation factor) of the table is < c

- At most cM items in total
  - 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 (max load=0.5) | 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 - YTJMAZ

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

# Outline

1. Introduction
2. Binary Search Tree
3. AVL Tree
4. Hash tables
   A. Introduction
   B. Chaining
   C. Probing
      I. Linear Probing

# Probing

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

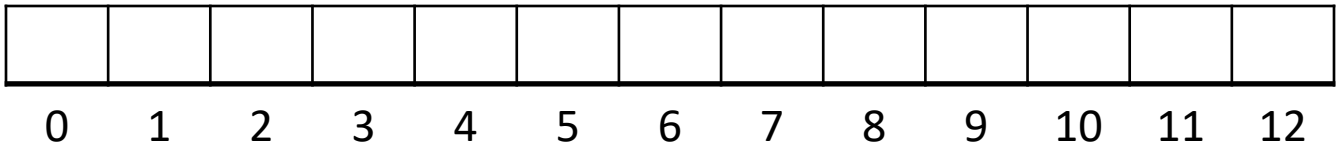
# Linear Probing

- 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]
```

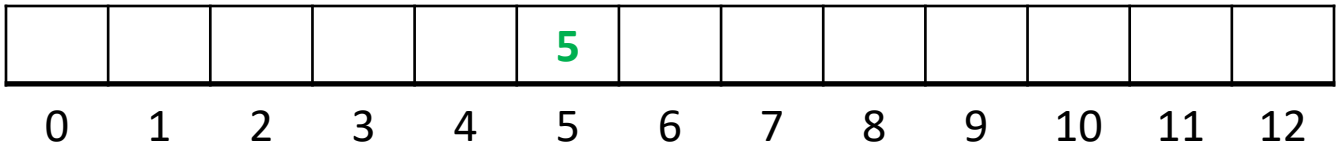


# Linear Probing

- 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]
```

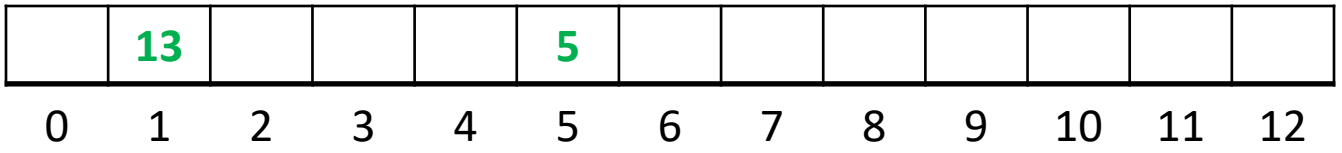


# Linear Probing

- 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]
```

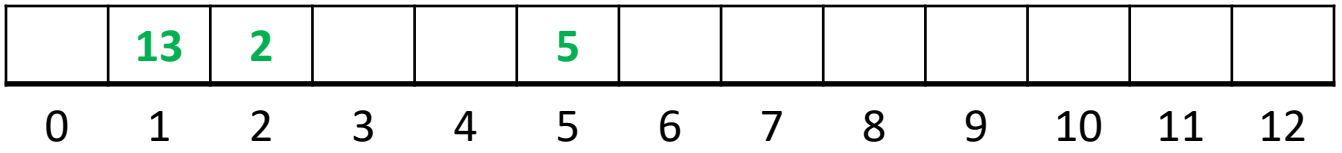


# Linear Probing

- 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]
```

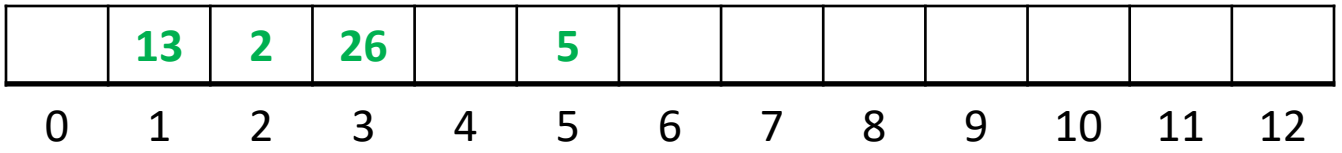


# Linear Probing

- 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]
```

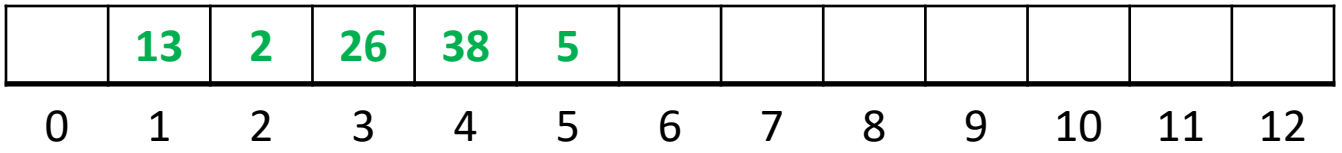


# Linear Probing

- 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?**

- All Elements in one cluster
- This degenerates to linear seach: O(M)

**Resizing**

- In general, we resize (double) the table size once the **load factor** exceeds a certain value (different for different hashing schemes)
- Load factor  c = number of elements in table / size of table
- This means the table is never full
- Still O(c*M) = O(M) for search

# Linear Probing

**Deletion:**

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

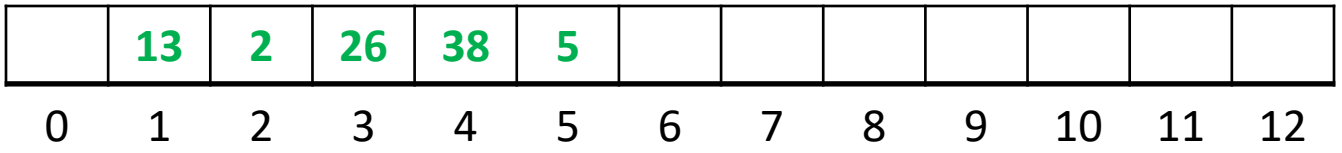
# Linear Probing

**Deletion:**

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

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

- Delete 2



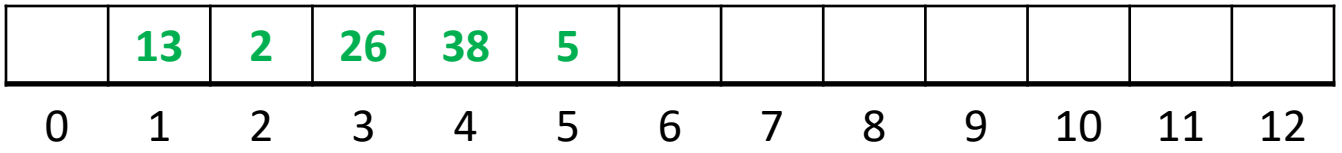
# Linear Probing

**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



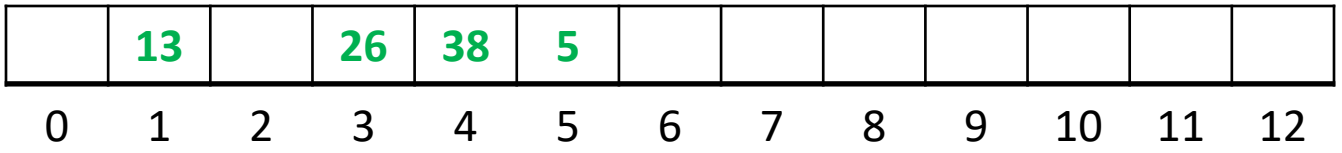
# Linear Probing

**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



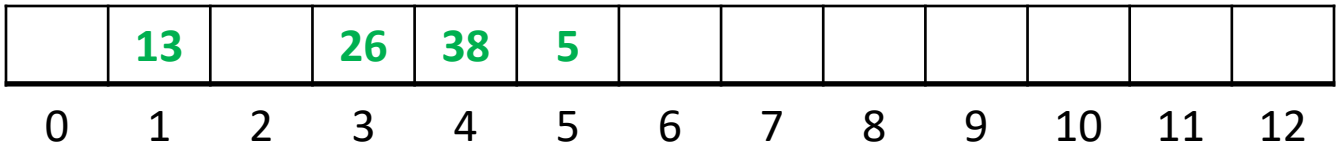
# Linear Probing

**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?



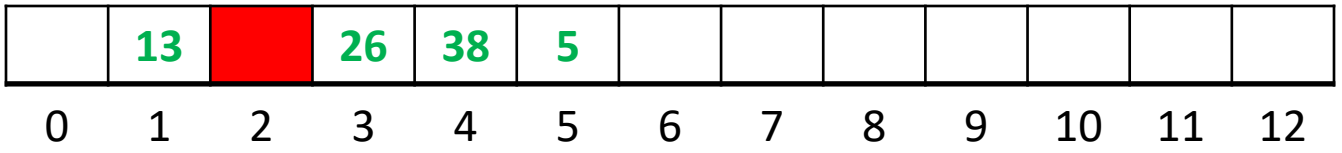
# Linear Probing – Deletion

**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

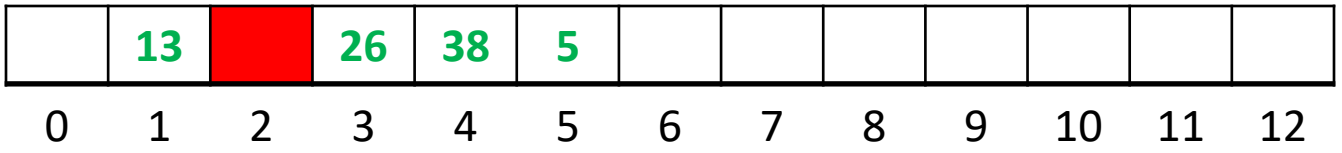
**Deletion:**

- 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!

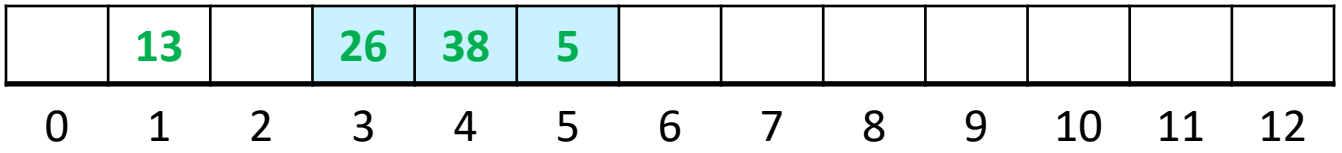
Quiz time!
https://flux.qa - YTJMAZ



# Linear Probing – Deletion 1

**Deletion:**

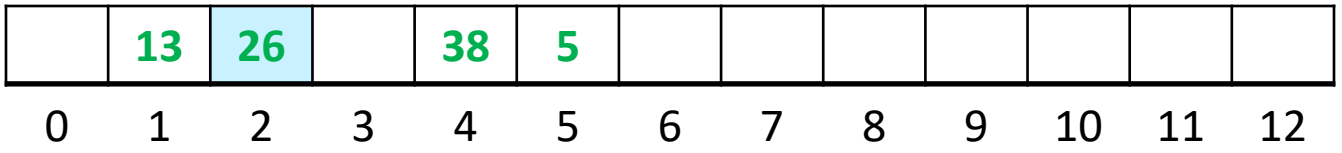
- Search the element
- Delete it
- Is that all?
- Solution: Re-insert everything after the deleted position, until the next empty position



# Linear Probing – Deletion 1

**Deletion:**

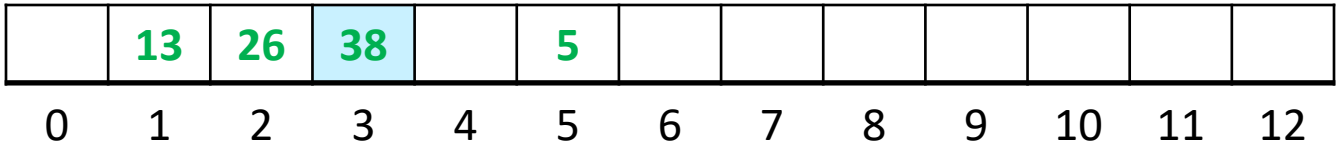
- Search the element
- Delete it
- Is that all?
- **Solution: Re-insert everything after the deleted position, until the next empty position**
- Insert 26



# Linear Probing – Deletion 1

**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



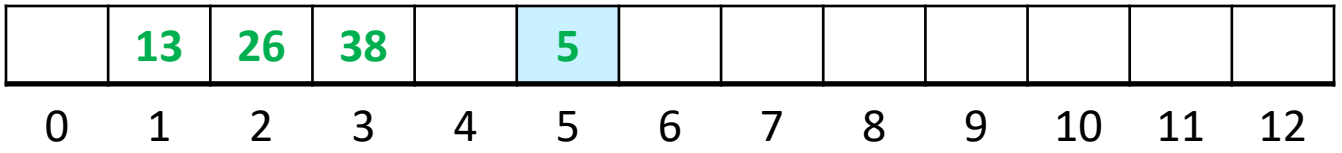
# Linear Probing – Deletion 1

**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

- **Are we finished now?**

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

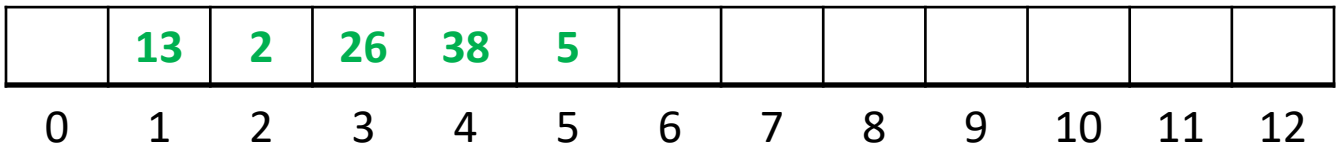


# Linear Probing – Deletion 2

**Deletion:**

- Search the element
- Mark it as deleted (“tombstone”)
- When searching, treat marked elements as regular elements for probing purposes
- If searching for an element, and it is marked, overwrite it

- Delete 2

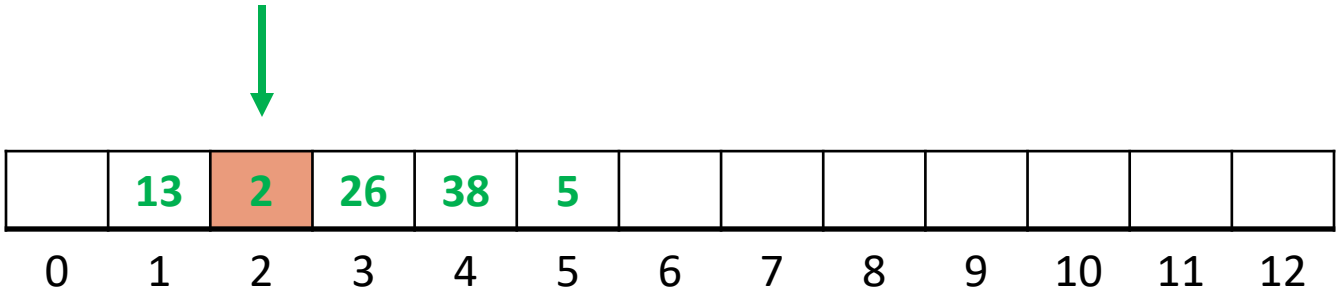


# Linear Probing – Deletion 2

**Deletion:**

- Search the element
- Mark it as deleted
- When searching, treat marked elements as regular elements for probing purposes
- When inserting, if you probe to a marked element, you treat it as an empty space (i.e. you overwrite it with the value you are inserting)

- Delete 2
- Search for 26 - **h(k,i) = k%12 + I**

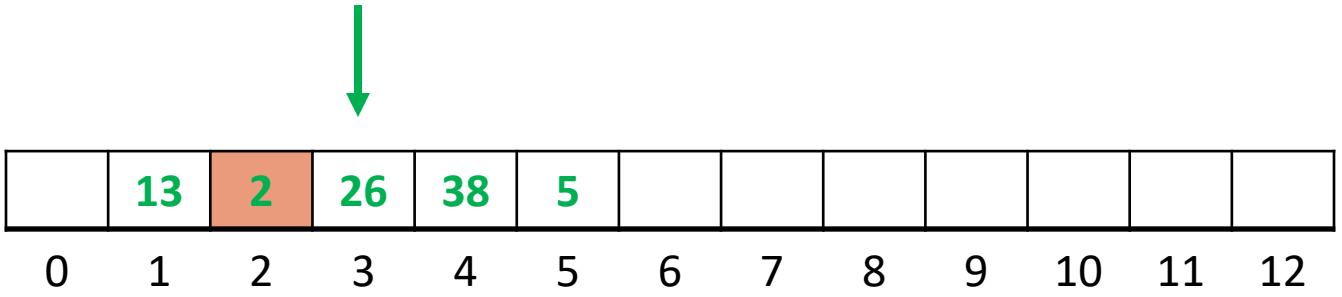


# Linear Probing – Deletion 2

**Deletion:**

- Search the element
- Mark it as deleted
- When searching, treat marked elements as regular elements for probing purposes
- If searching for an element, and it is marked, overwrite it

- Delete 2
- Search for 26 - **h(k,i) = k%12 + I**

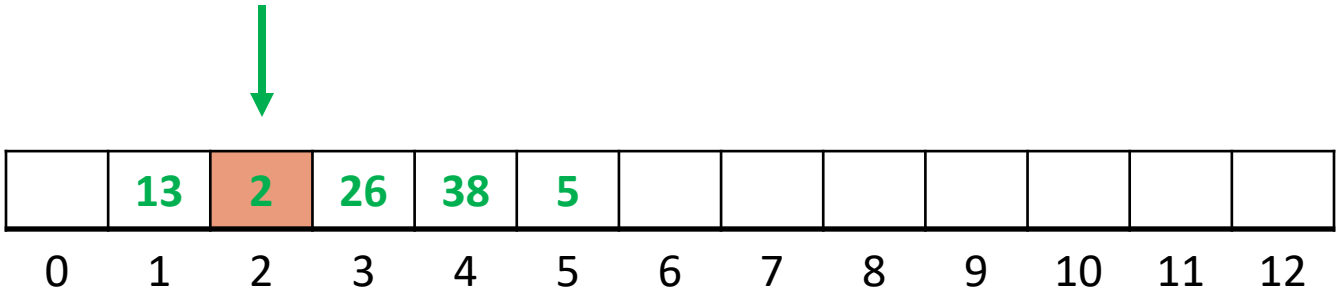


# Linear Probing – Deletion 2

**Deletion:**

- Search the element
- Mark it as deleted (this called “Lazy Deletion”)
- When searching, treat marked elements as regular elements for probing purposes
- If searching for an element, and it is marked, overwrite it

- Delete 2
- Search for 26 - **h(k,i) = k%12 + I**
- Insert 14



# Linear Probing – Deletion 2

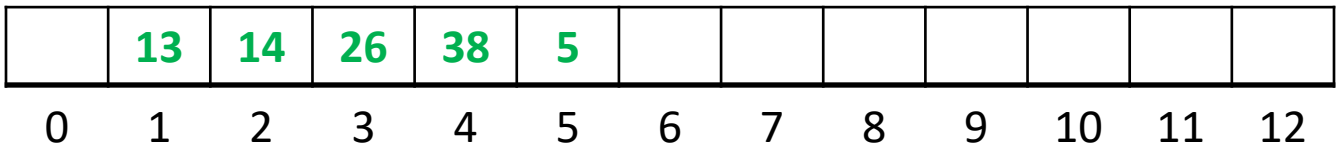
**Deletion:**

- Search the element
- Mark it as deleted
- When searching, treat marked elements as regular elements for probing purposes
- If searching for an element, and it is marked, overwrite it

- Delete 2
- Search for 26 - **h(k,i) = k%12 + I**
- Insert 14

- **Are we finished now?**

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



# 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 clustering.

(more precisely, *primary* clustering. *Secondary* clustering occurs with more complex types of probing)

| | 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. Improved hashing [Quadratic Probing, Double and Cuckoo Hashing]
5. Randomised Lookup (Teaser)

# Randomised Alternatives (non examinable)

- Randomised data structures replace the deterministic operations we have used so far by randomized ones.
- They can often be a great alternative (simple & efficient) to deterministic structures
- However, due to the probabilistic nature their efficiency can usually only their average efficiency can be assessed
- A great example are Skip-Lists
  Pugh, W. (1990). "Skip lists: A probabilistic alternative to balanced trees" Communications of the ACM. 33 (6): 668–676.

# Skip Lists (non examinable)

"Parallel" linked lists with "Express lanes"

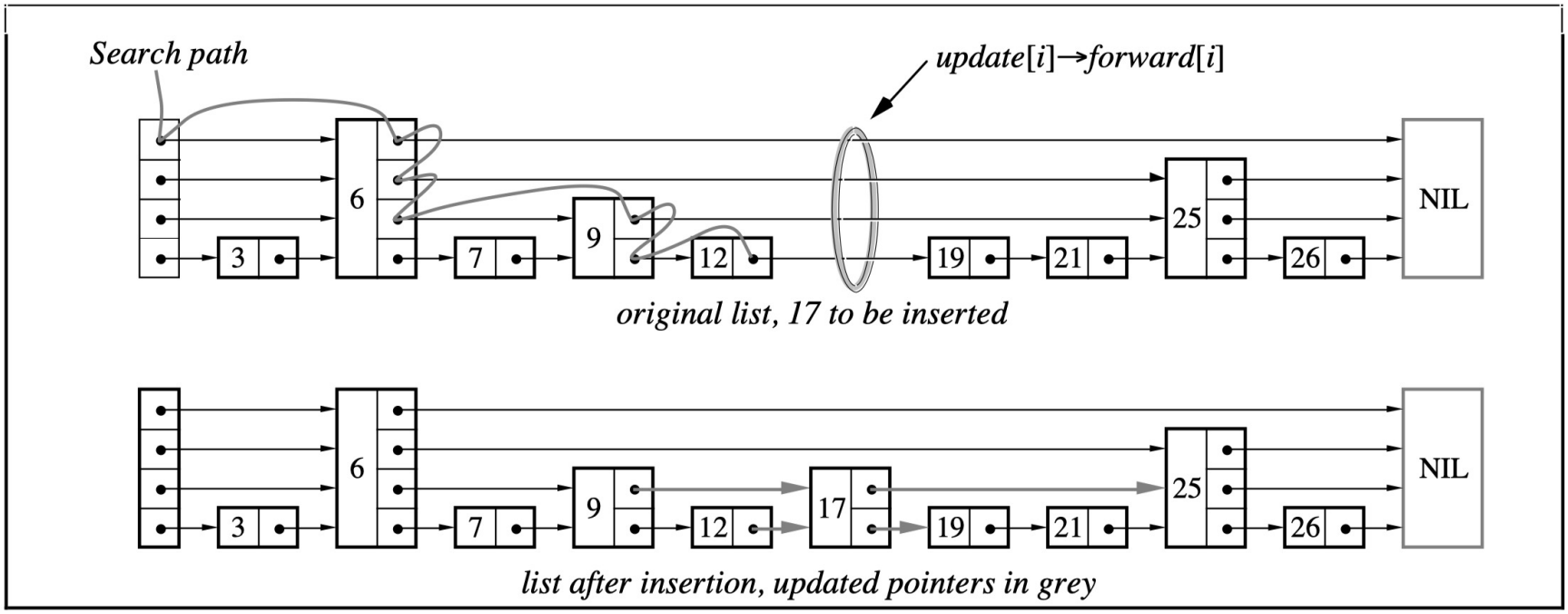



FIGURE 3 - Pictorial description of steps involved in performing an insertion

# Skip Lists (non examinable)

When inserting a new element ("tower") determine the height of the tower by repeatedly flipping a coin: "increase one more level?"

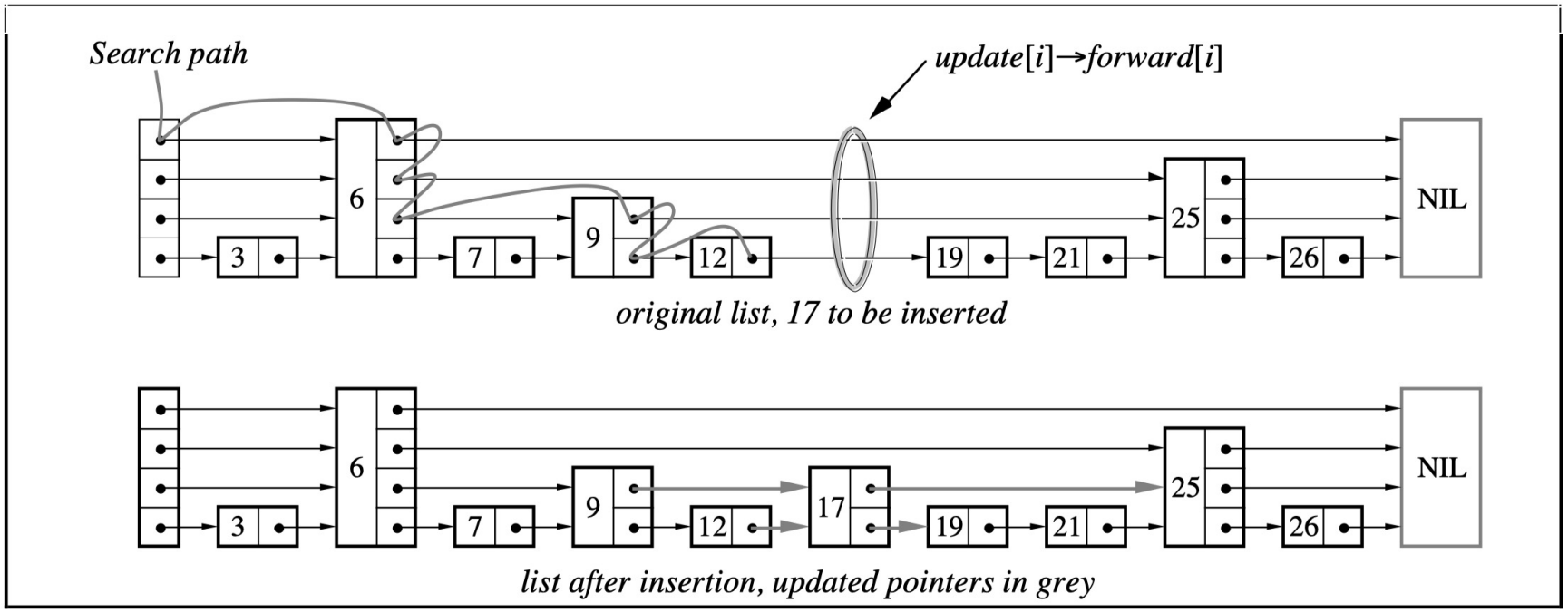



FIGURE 3 - Pictorial description of steps involved in performing an insertion

# Skip Lists (non examinable)

Average (expected) time complexity for all operations is O(log n)

- Dominate by search. Insert/Delete in linked list is O(1)

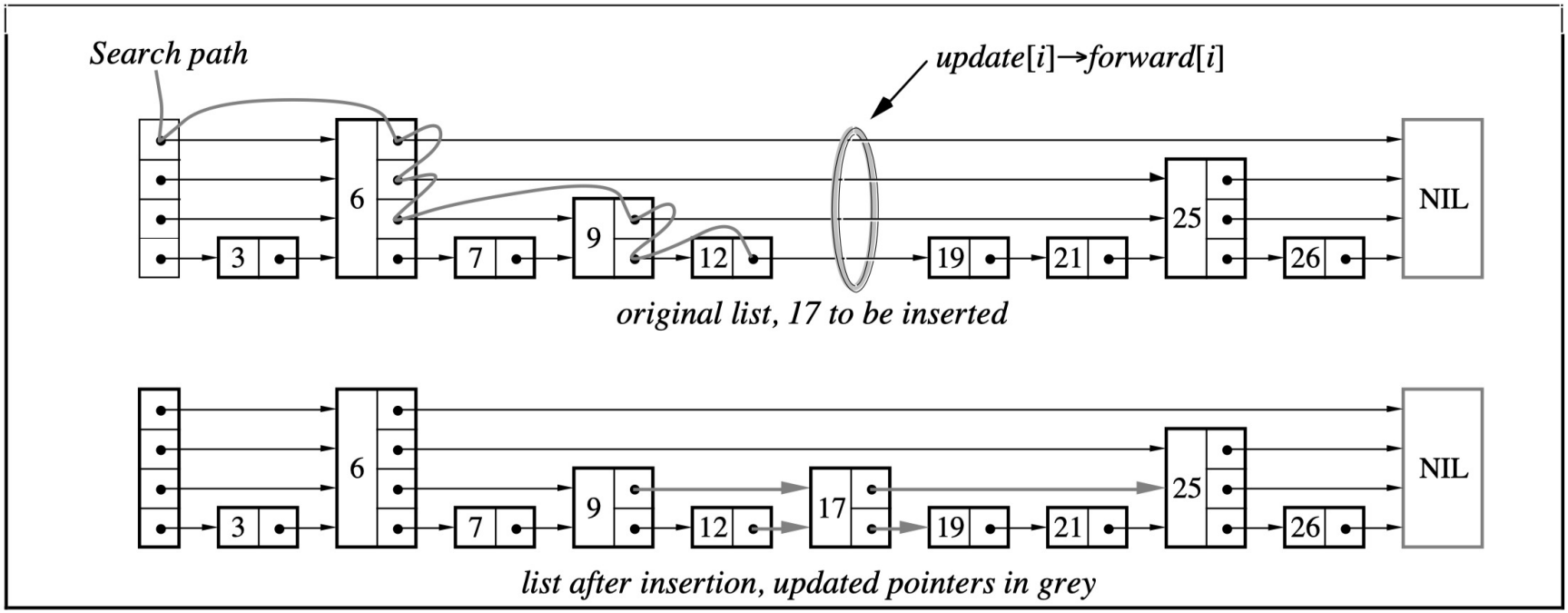



FIGURE 3 - Pictorial description of steps involved in performing an insertion

# Summary

## Take home message

- Hash tables provide O(1) look up in practice
  - but 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
- Read and implement Skip lists

## Coming Up Next

- Retrieval Data Structures for Strings