Faculty of Information Technology, Monash University

FIT2004, S2/2016

Week 5: Efficient Lookup Structures

Lecturer: Muhammad Aamir Cheema

ACKNOWLEDGMENTS

The slides are based on the material developed by Arun Konagurthu and Lloyd Allison.

Announcements

- Assessed prac sheet for week 6 released
  - Deadline: Monday, 29-Aug-2016 10:00:00 **AM**
  - Submissions will be passed through MOSS to detect plagiarism
- Programming Competition: Round 1 closes end of next week
- Fill in the anonymous survey if you have not already

Overview

- Hash Tables
- Binary Search Tree
- AVL Tree

Recommended Reading

- Hashing: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Table/
- Search Trees part of http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/
- Weiss “Data Structures and Algorithm Analysis in Java" (Chapter 5 and Chapter 4: Sections 4.1-4.4)

Lookup Table

- 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 identifies unambiguously an element
- Elements can be looked up (or searched) using this key. (Note, the element with no extra data/attributes is itself the key)
- Elements of a table, in some cases, contain a key plus some other attributes or data

Sorting based lookup

Keep the elements sorted on their keys in an array

Insertion:

- 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)

Searching:

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

Deletion:

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

Is it possible to do better?

Yes! hash tables and AVL trees!

Direct-Addressing

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

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

Searching the record with ID x

- Return array[x]

Note that this is similar to the Task 1A in week 4 assessment

Problem with Direct-Addressing

- We assumed the keys (e.g., ID) range from 1 to N
- What if this is not true?
  - IDs are not sequentially numbered (e.g., 26787973, 3167814 etc.)
  - Key is authcate (e.g., alpha3, beta5 etc.)

Direct-Addressing is not suitable if the Universe of the keys is large

Fixing the Problem with Direct-Addressing

Assume that we need to store the records for N students in a way to allow efficient lookup

Idea:

- Create an array of size N
- Store a student with key k at index k % N, e.g., if N is 10 and student ID is 26787973, store the student at index 26787973 % 10 = 3
  - If the key is a string (e.g., authcate), convert it to a number k (e.g., using ASCII values) and then store at index k % N

Problem

- Two students may get the same index (e.g., 26787973 % 10 = 3 and 31678143 % 10 = 3)

The above is a very simplified idea behind hashing. The problem discussed above is called *collision*.

Hashing

- A hash function maps key values onto an index position in an array of elements, i.e., index = hash(k) where hash() denotes the hash function.
- The array is used as an implementation of the hash table.
- Elements of the table can then be accessed directly using the hash key → hash index transformation, and then looking up the array at the position pointed by hash index. That is, hash index is the array index.
- A problem with hashing is collisions – when two or more keys are mapped to same index position.
- Can we avoid collisions by defining better hash functions?

Understanding hash functions

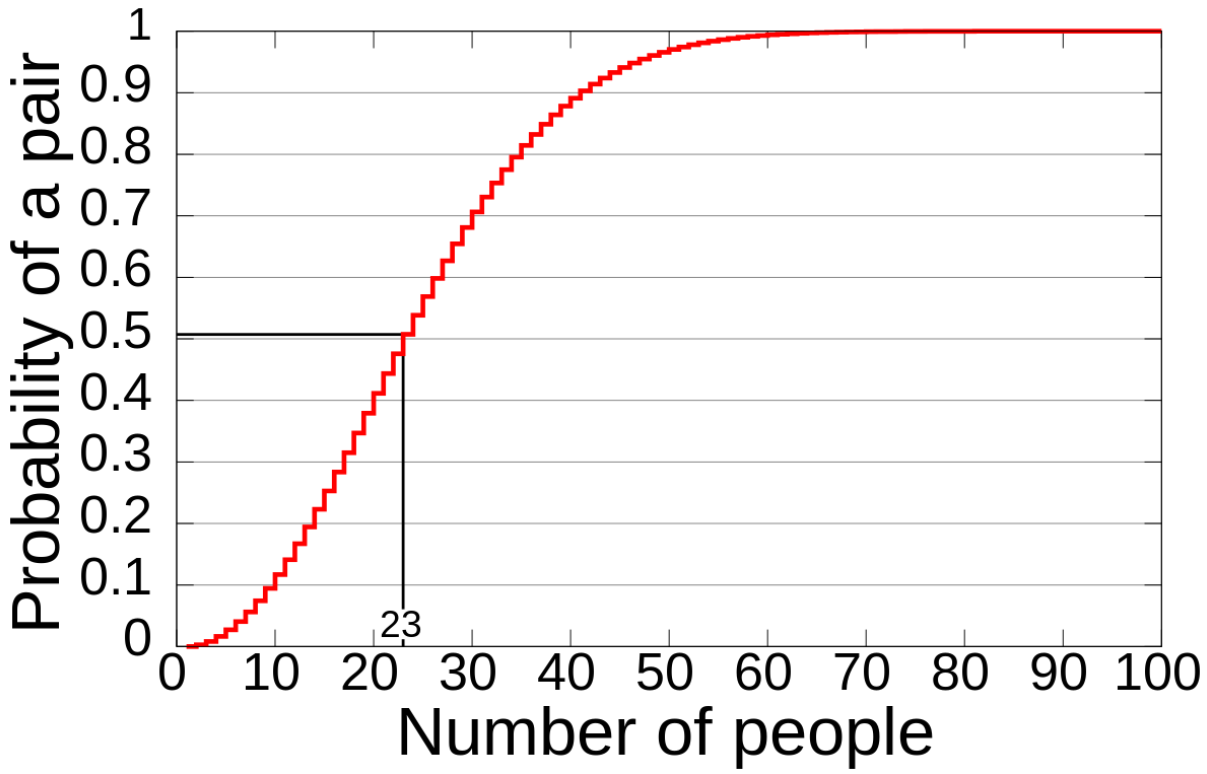
- We want to use a hash table for a class of 50 students
- Assume that the hash function is based on birthdays (dd-mm), e.g., a student born on 01-Jan is hashed to index 1, 02-Jan on index 2, …, 31-Dec on 365.
- How likely is that two students will be hashed to the same index, e.g., how likely is the collision?

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

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

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%



Take home message for hash functions

In this birthday paradox exercise we conducted in the class:

- We considered a hash table was an array of size M equal to 365.
- The birth date dd-mm is the key. A hash function f(dd-mm) mapped dd-mm into a number/index d between [1 … 365].
- As the number of people (N) grows, the hash index f(dd-mm) has an increasing probability of collision.
- The seemingly counterintuitive part was that N was NOT very large (in comparison with the table size M) for collisions to occur -- this is generally true for most hash functions!

In short

- Given N keys and a hash table size of M, collisions will always occur unless M >> N
- In plain English: Cannot avoid collisions unless we decide to use a ridiculous amount of memory to store the hash table!

Some facts about hash functions

- It is impossible to design a good hash function for all circumstances.
- In general, designing good hash functions requires analysing the statistical properties of the key type.
- The ideal hash function maps the actual key values uniformly onto the hash table indexes - Unfortunately the ideal is almost always unrealizable!
- The surprising part is that the best hash functions are essentially pseudo-random functions.
- However, collisions from these hash functions are, in almost all practical cases, inevitable.

Handling Collisions

- Now, we know that collisions are unavoidable.
- How do we address collisions
  1. Separate chaining or open hashing or closed addressing
  2. Closed hashing or open addressing

Separate Chaining

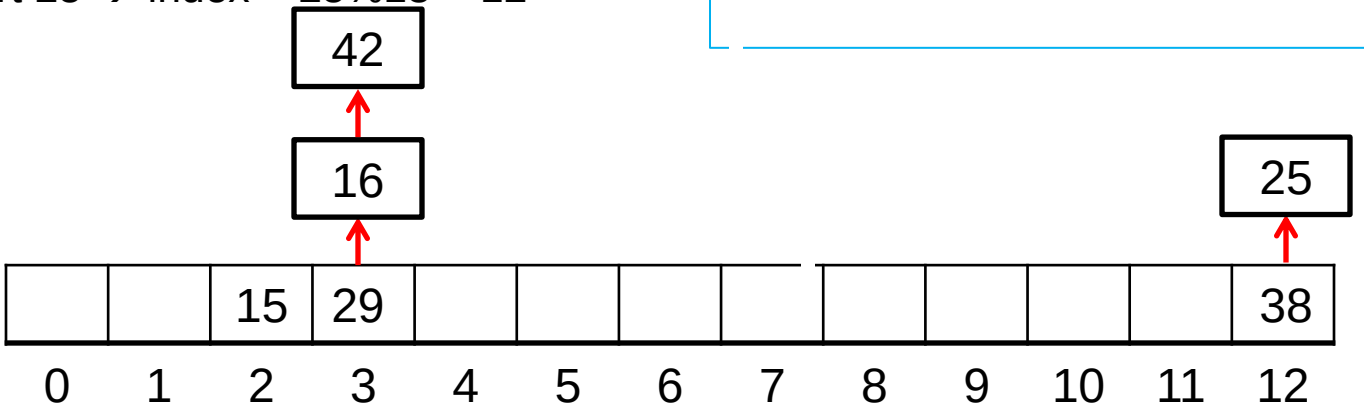
- If there are already some elements at hash index
  - Add the new element in the list at array[index]
- Example: Suppose the hash table size M is 13 and hash function is key % 13.
  - Insert 29 → index = 29%13 = 3
  - Insert 38 → index = 38%13 = 12
  - Insert 16 → index = 16%13 = 3
  - Insert 15 → index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



Separate Chaining

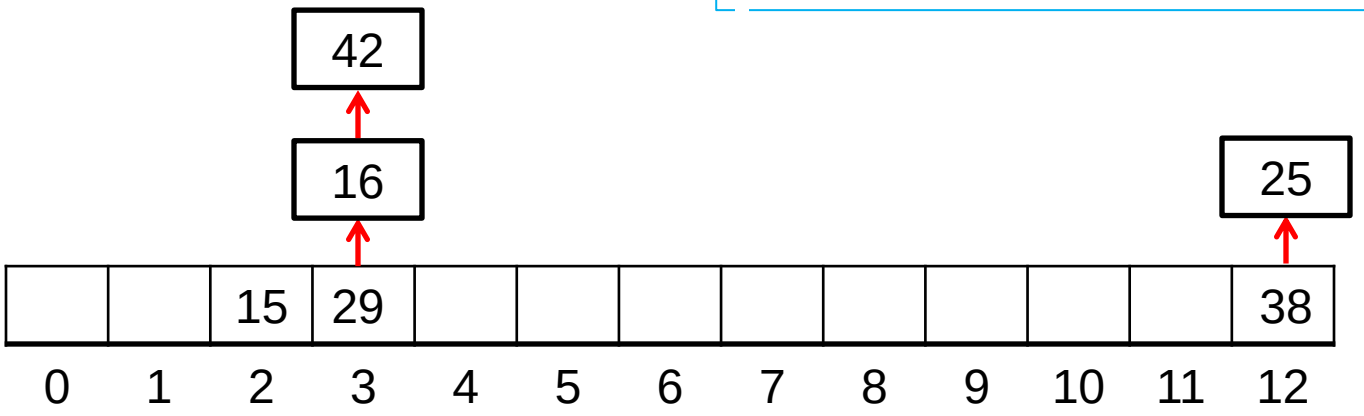
- Best-case Time complexity
  - Inserting an element
  - Searching an element
  - Deleting an element
- Worst-case Time complexity
  - Inserting an element
  - Searching an element
  - Deleting an element

What if we use sorted array instead of a linked list?

Worst-case Time complexity:

- Insertion
- Search
- Deletion

We could also use other structures such as AVL trees instead of linked list.



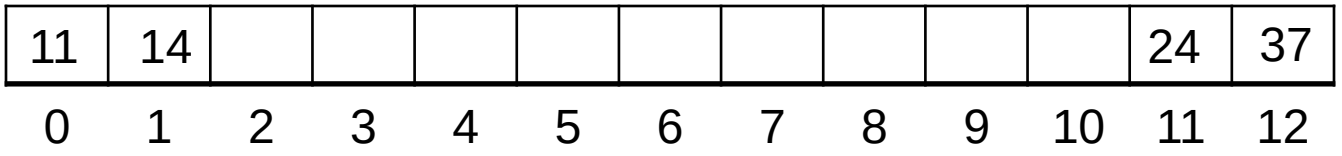
Closed Hashing (Open Addressing)

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

Linear Probing

- In case of collision, sequentially look at the next indices until you find an empty index OR return fail if the hash table is full.
- Example:
  - Insert 24 → index = 24 % 13 = 11
  - Insert 14 → index = 14 %13 = 1
  - Insert 37 → index = 37 % 13 = 11
    - Oops! Index 11 is occupied
    - Insert it at next index (11 + 1)%13 = 12
  - Insert 11 → index = 11 % 13 = 11
    - Oops! Index 11 is occupied
    - Check next index → (11 + 1)% 13 = 12
    - Oops! Index 12 is occupied
    - Check next index → (11 + 2)%13 = 0
    - Insert at index 0

```
// 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 = hash(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)
- or you have scanned all indices (which implies that the element is not in the hash table)

Example: Search 53: (Index = 53 % 13 = 1)
Search 27: (Index = 27 % 13 = 1)

Deletion:

- Search the element
- Delete it
- AND set array[index] = DELETED // This is important! Why?

Example:

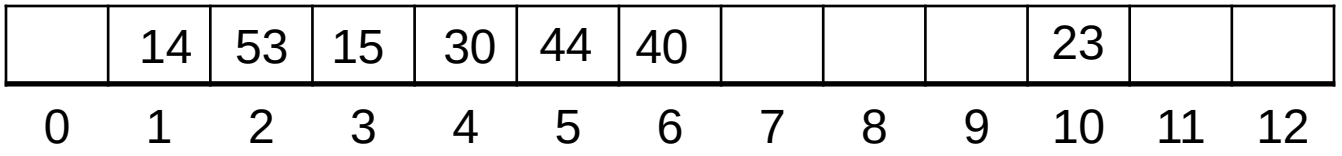
Insert 40 in the array.
Delete 15 from the array
Search 40!
If the cell is not marked DELETED, the algorithm will return **not found** because array[3] is NIL.

	14	1	15	30	53	40				23		
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

- The previous example showed a search by increment of 1
- In general, we can sequentially search with increment of a constant c, e.g., (hash(key) + 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.



Quadratic Probing

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

- Linear probing: index = (hash(key) + c*i) % M
- Quadratic probing: index = (hash(key) + c*i + d*i^2) % M where c and d are constants

E.g., assume c = 0.5, d = 0.5

insert 40 → hash(40) = 40 % 13 = 1

i = 0 → index = 1 % 13 = 1

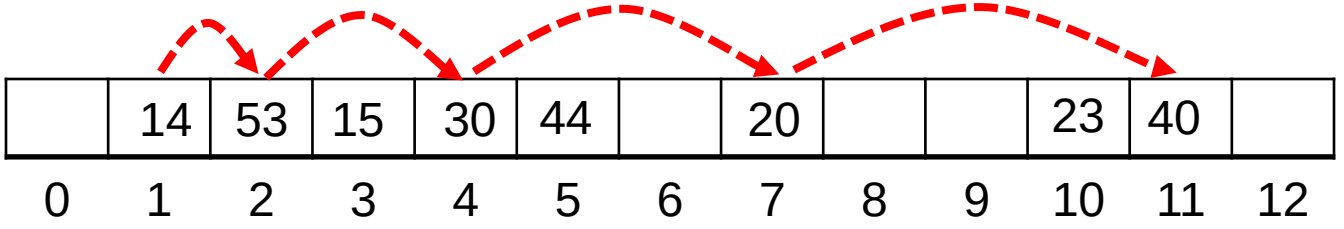
i = 1 → index = (1 + 0.5 + 0.5) % 13 = 2 // a jump of 1

i = 2 → index = (1 + 0.5*2 + 0.5*4) % 13 = 4 // a jump of 2

i = 3 → index = (1 + 0.5*3 + 0.5*9) % 13 = 7 // a jump of 3

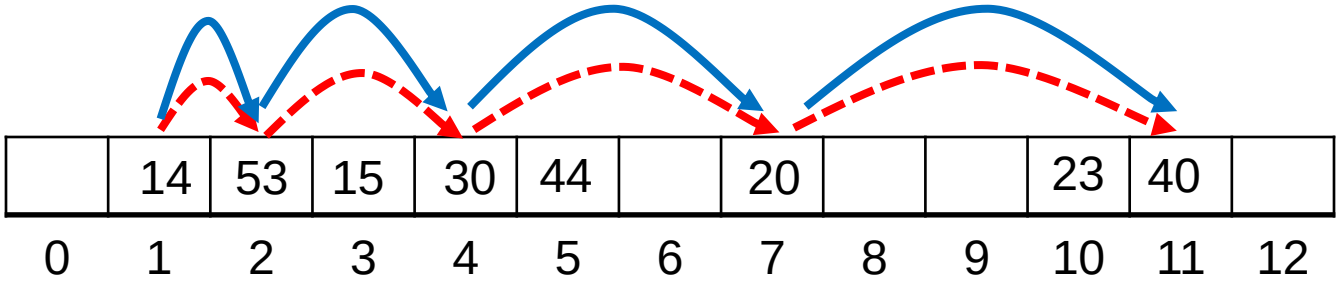
i = 4 → index = (1 + 0.5*4 + 0.5*16) % 13 = 11 // a jump of 4

- Quadratic probing is not guaranteed to probe every location in the table - an insert could fail while there is still an empty location. However, hash tables are rarely allowed to get full (to get good performance).
- The same probing strategy is used in the associated search and delete routine!



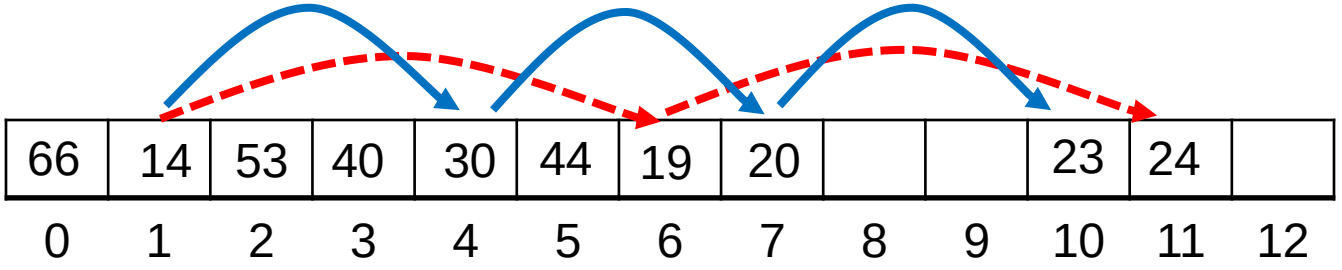
Problem with Quadratic Probing

- Quadratic probing avoids primary clustering
- However, if two elements have same hash index (e.g., hash(k1) = hash(k2)), the jumps are the same for both elements.
  - index = (hash(key) + c*i + d*i^2) % M
  - E.g., hash(40) = hash(66) = 1
  - dashed red arrows show jump for inserting 40 (as in previous slides)
  - Blue arrows show jumps for inserting 66
- 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



Double hashing

- Use two different hash functions: one to determine the initial index and the other two determine the amount of jump
- Index = (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



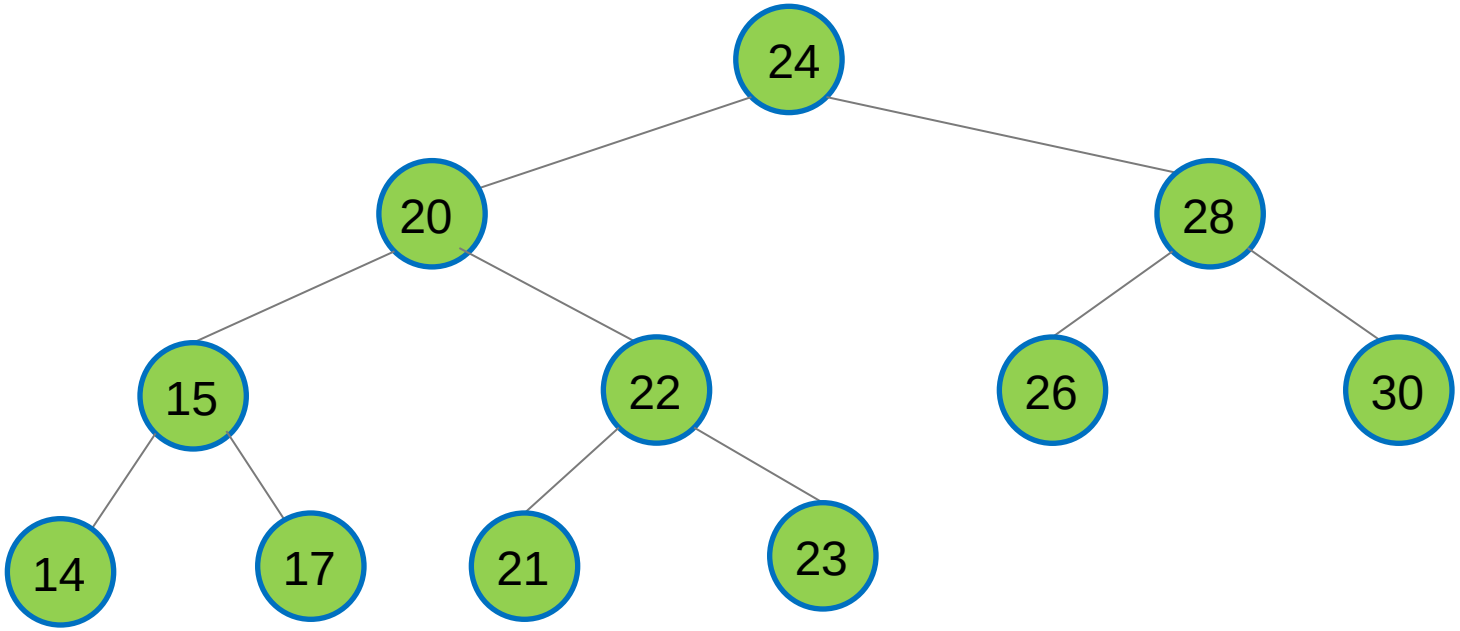
Summary of Hashing

- It is hard to design good hash functions.
- The examples shown in the lecture show very simple hash functions.
- Worst-case time complexity: O(N)
- In practice hash tables give quite good performance (e.g., O(1))
- Hash tables are disordered data structures. Therefore certain operations become expensive.
  - Find maximum and minimum of a set of elements (keys).

Binary Search Tree (BST)

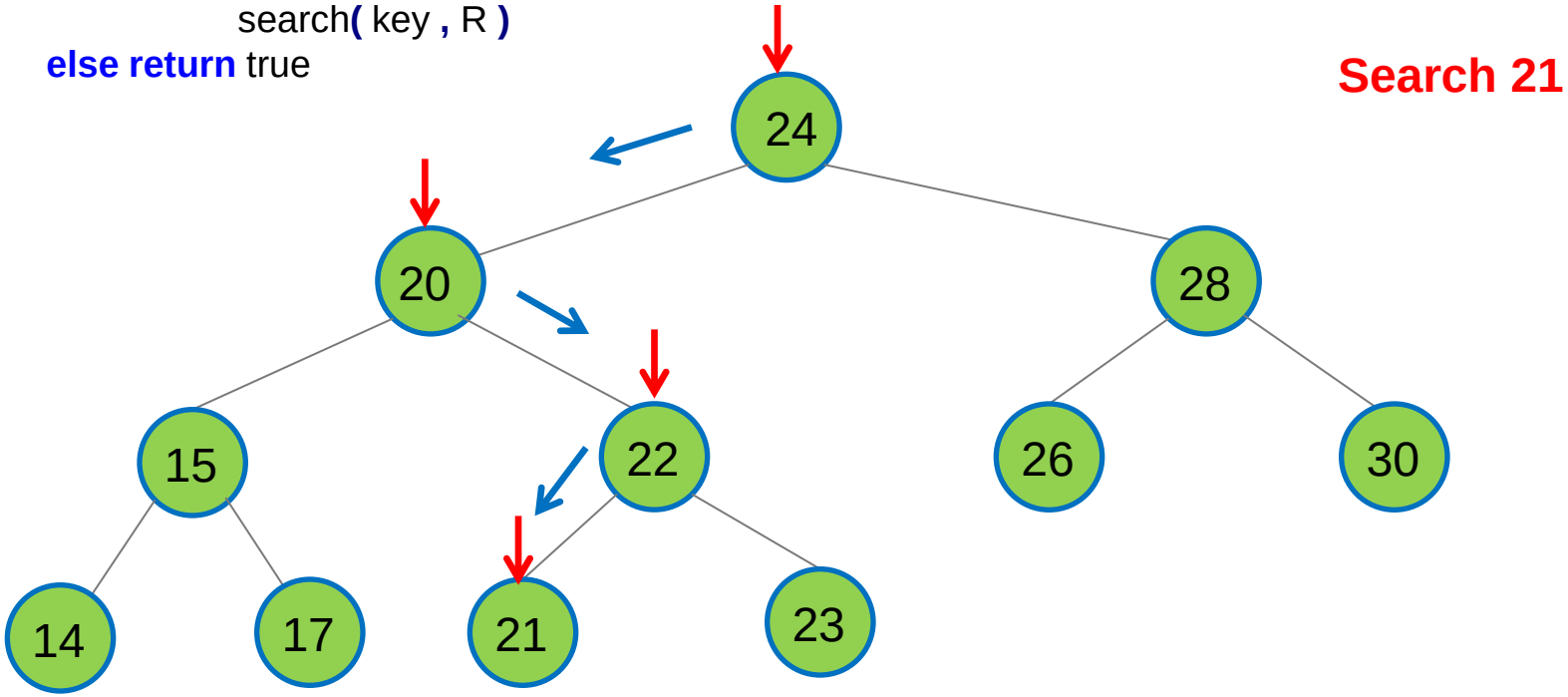
- The empty tree is a BST
- If the tree is not empty
  1. the elements in the left subtree are LESS THAN the element in the root
  2. the elements in the right subtree are GREATER THAN the element in the root
  3. the left subtree is a BST
  4. the right subtree is a BST

Note! Don't forget last two conditions!



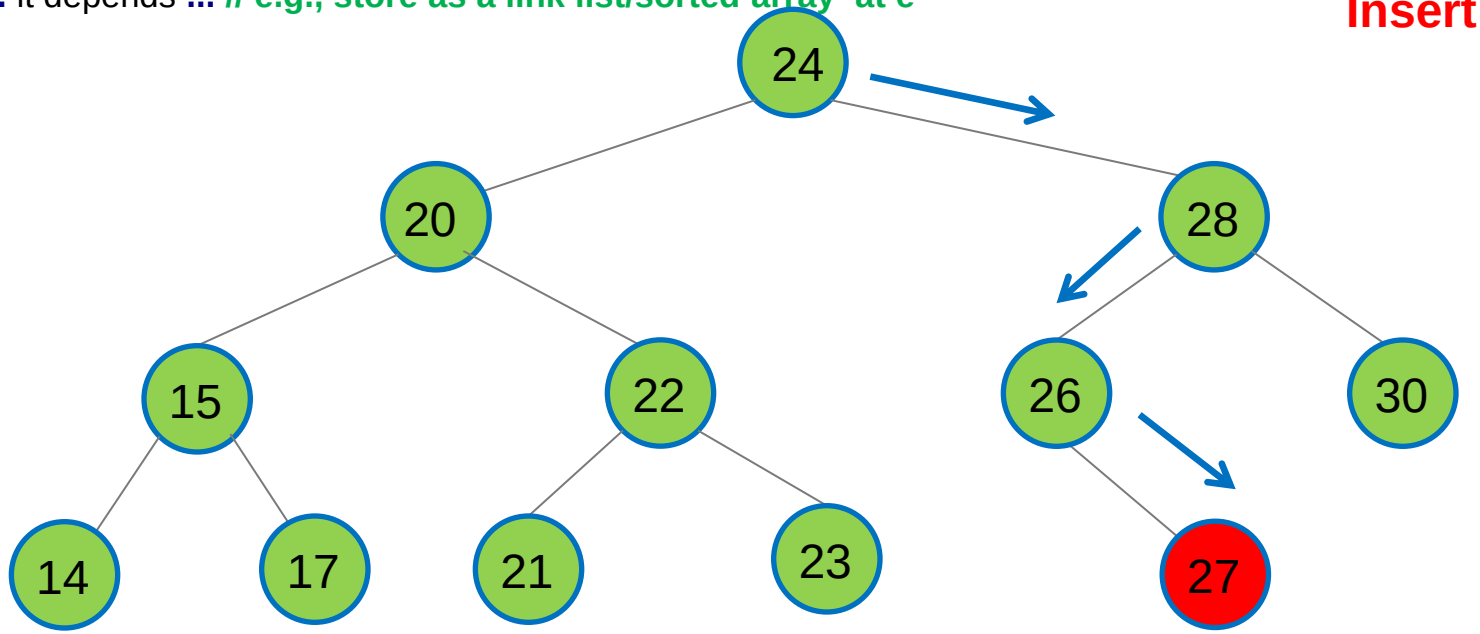
Searching a key in BST

```
// BST implemented here as a tree data structure
// T = fork(e, L, R)
function search(key,T)
        if (T == nilTree)
                return false // not present!
        else if ( key < e ) // search x in Left subtree
                search( key , L )
        else if ( key > e ) // search x in Right subtree
                search( key , R )
        else return true
```



Insert a key x in BST

```
// BST implemented here as a tree data structure
// T = fork(e, L, R)
function insert( x , T )
    if (T == nilTree) // Insert here as leaf node
        T = fork( x , nilTree , nilTree )
    else if ( x < e ) // Traverse and insert ...
        insert( x, L ); // along the Left subtree
    else if (x > e ) // Traverse and insert ...
        insert( x , R ) // along the Right subtree
    else // x == e
        ... it depends ... // e.g., store as a link list/sorted array  at e
    return
```



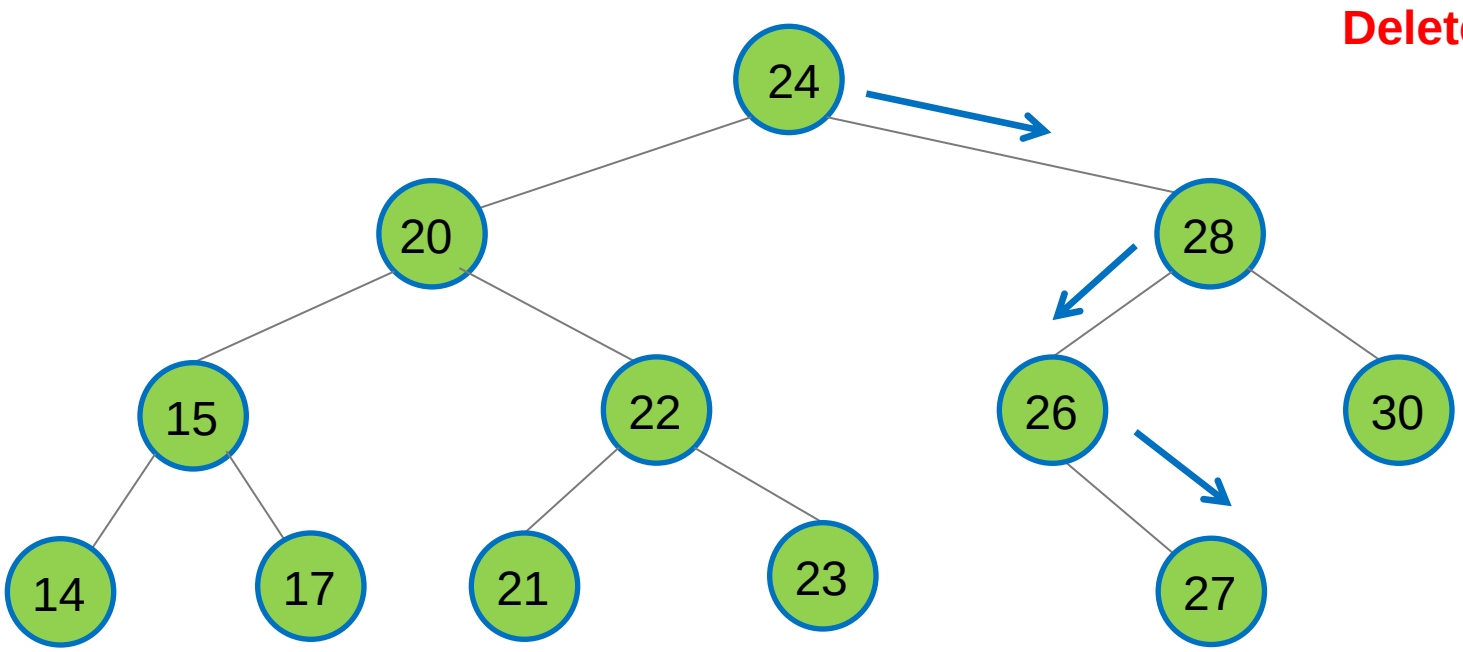
Delete a key from BST

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

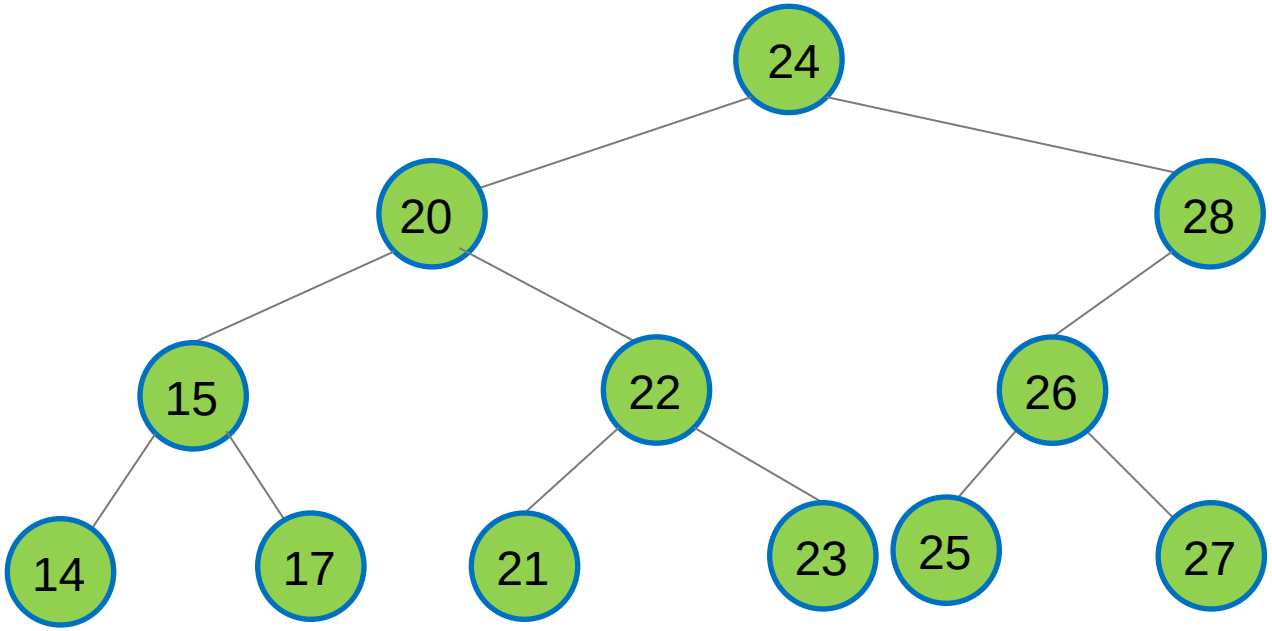
First lookup key in BST

If the key node has one child // Case 2

delete the key node

replace the key node with its child

Delete 28



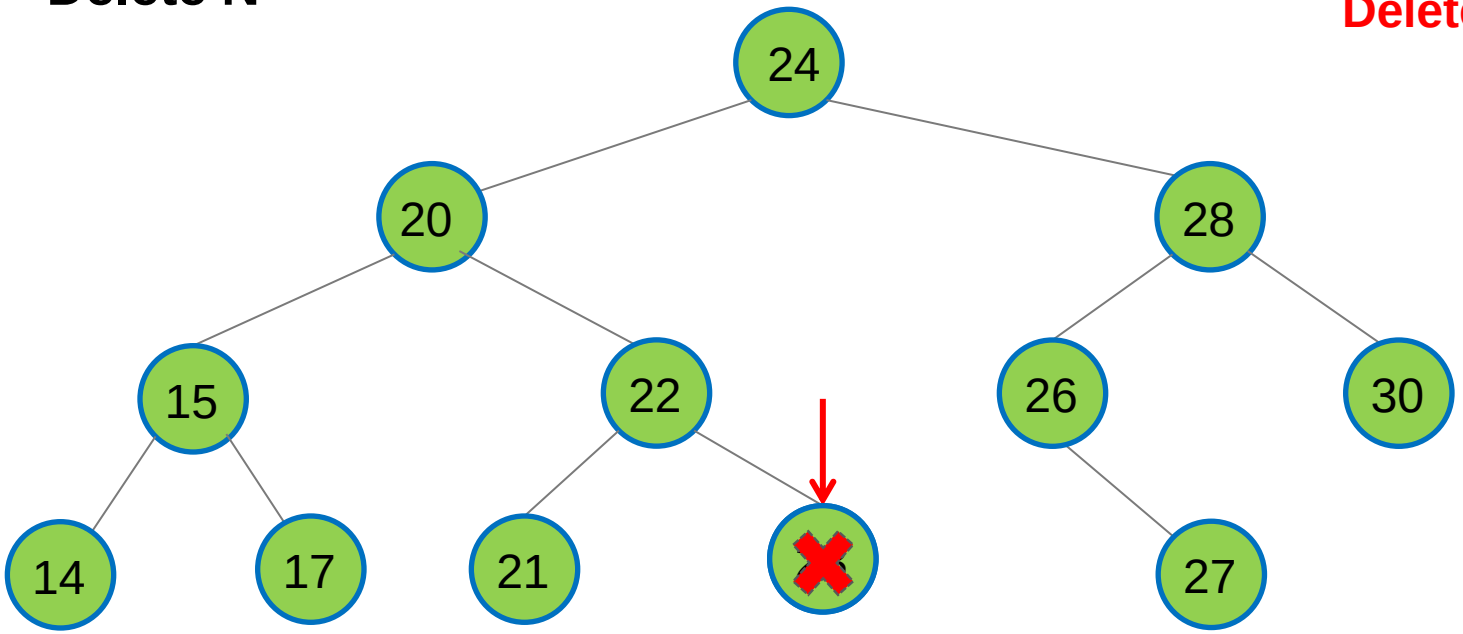
Delete a key from BST

First lookup key in BST

If the key node has two children // Case 3

- **Find the largest node N in left subtree (or the smallest node N in right subtree)**
- **Replace key node value with the value of N**
- **Delete N**

Delete 24



Delete a key from BST

First lookup key in BST

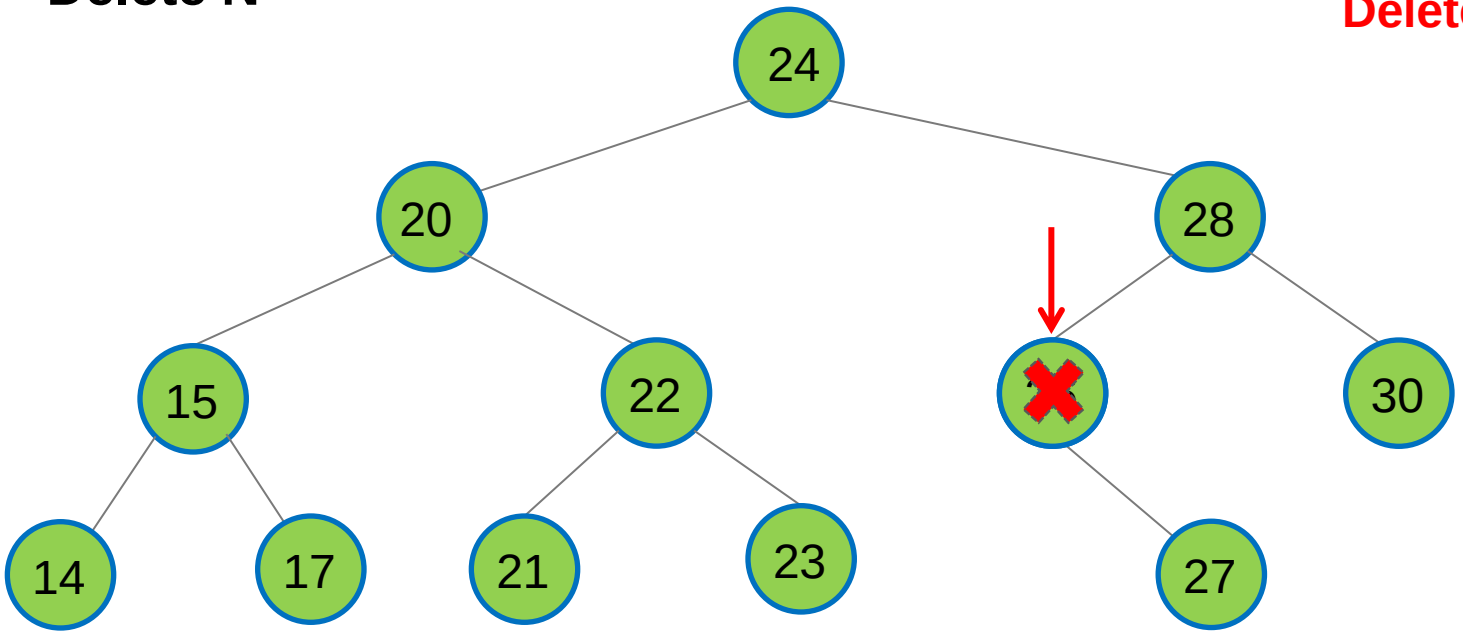
If the key node has two children // Case 3

Find the largest node N in left subtree (or the smallest node N in right subtree)

Replace key node value with the value of N

Delete N

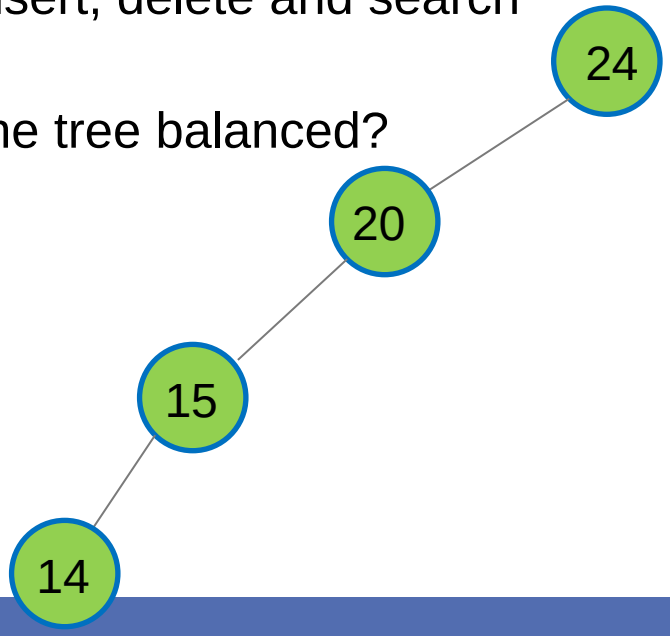
Delete 24



Worst-case of BST

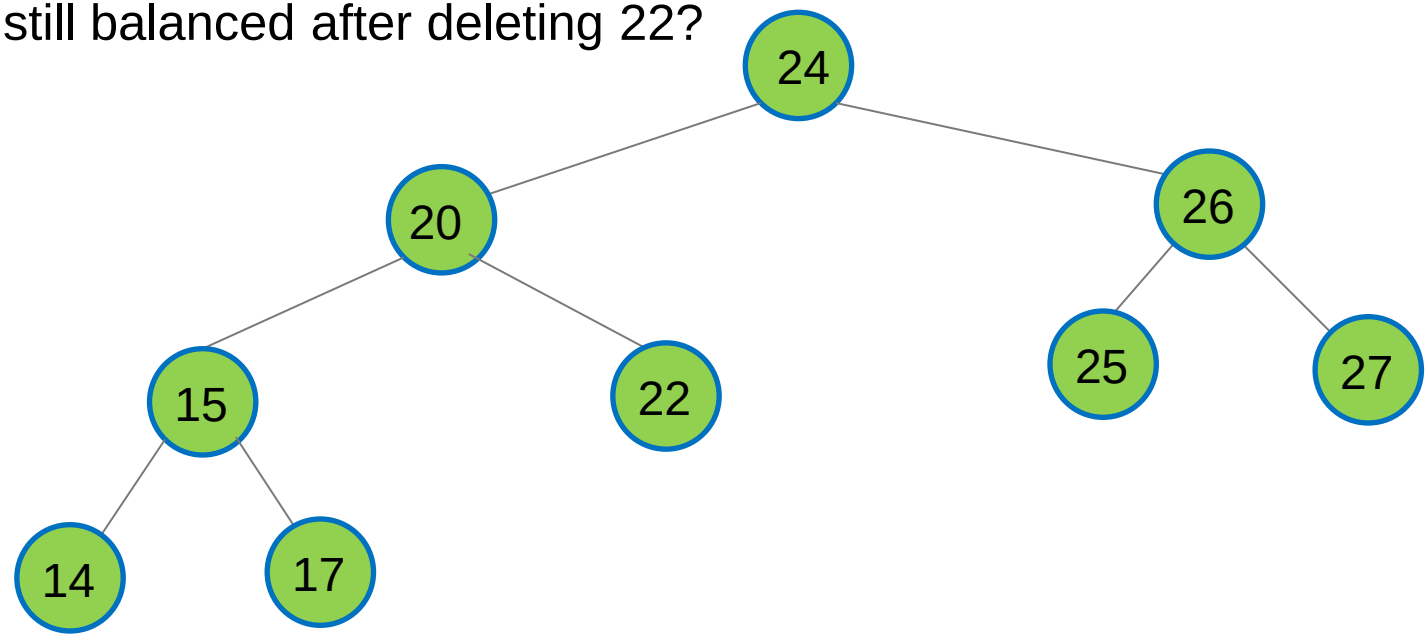
- A BST is not a balanced tree and, in worst case, may degenerate to a linked list
  - E.g., when elements are inserted in sorted order (ascending or descending) – insert 24, 20, 15, 14.
- Worst-case time complexity
  - Insert
  - Delete
  - Search
- Average-case time complexity is O(log N) for insert, delete and search
- Can we improve the performance by keeping the tree balanced?

Yes – AVL Tree does that



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?
- Is it still balanced after deleting 25?
- Is it still balanced after deleting 17?
- Is it still balanced after deleting 22?

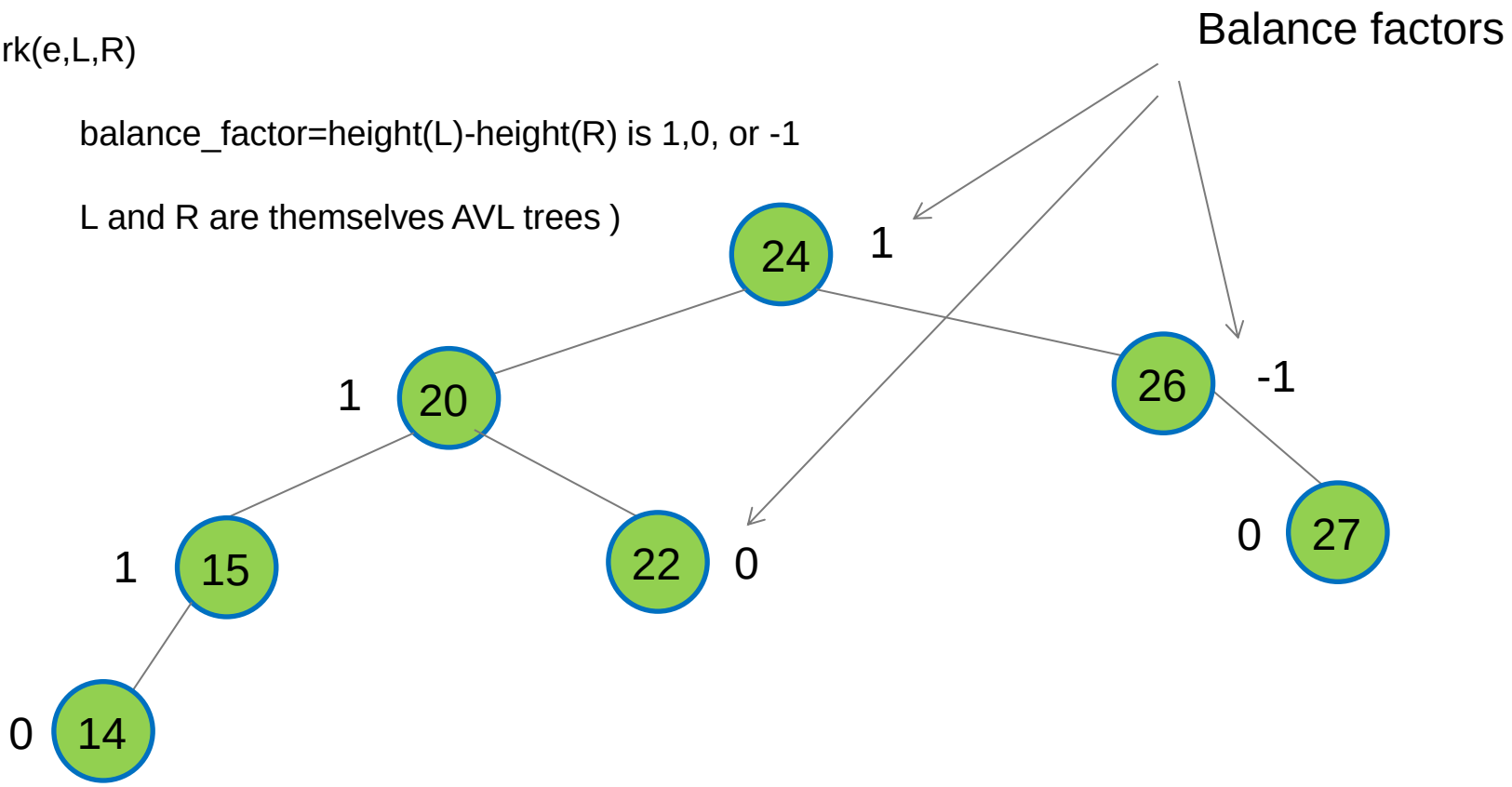


Defining AVL Tree

height(nilTree) = 0
height(fork(e,L,R)) = 1 + max(height(L), height(R))

Definition

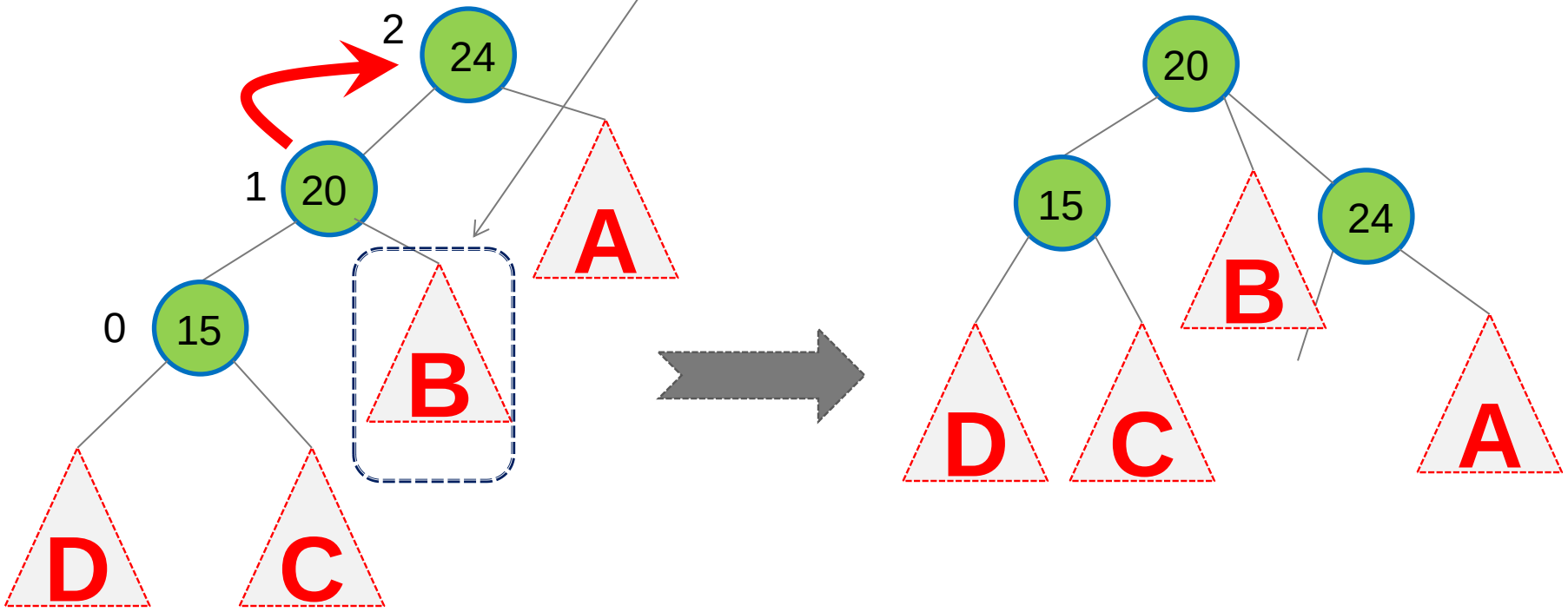
T is an AVL Tree if T is a binary search tree, and . . .
(T = nilTree)
OR
(T = fork(e,L,R)
AND
balance_factor=height(L)-height(R) is 1,0, or -1
AND
L and R are themselves AVL trees)



Keeping AVL Tree Balanced

Left Left Case

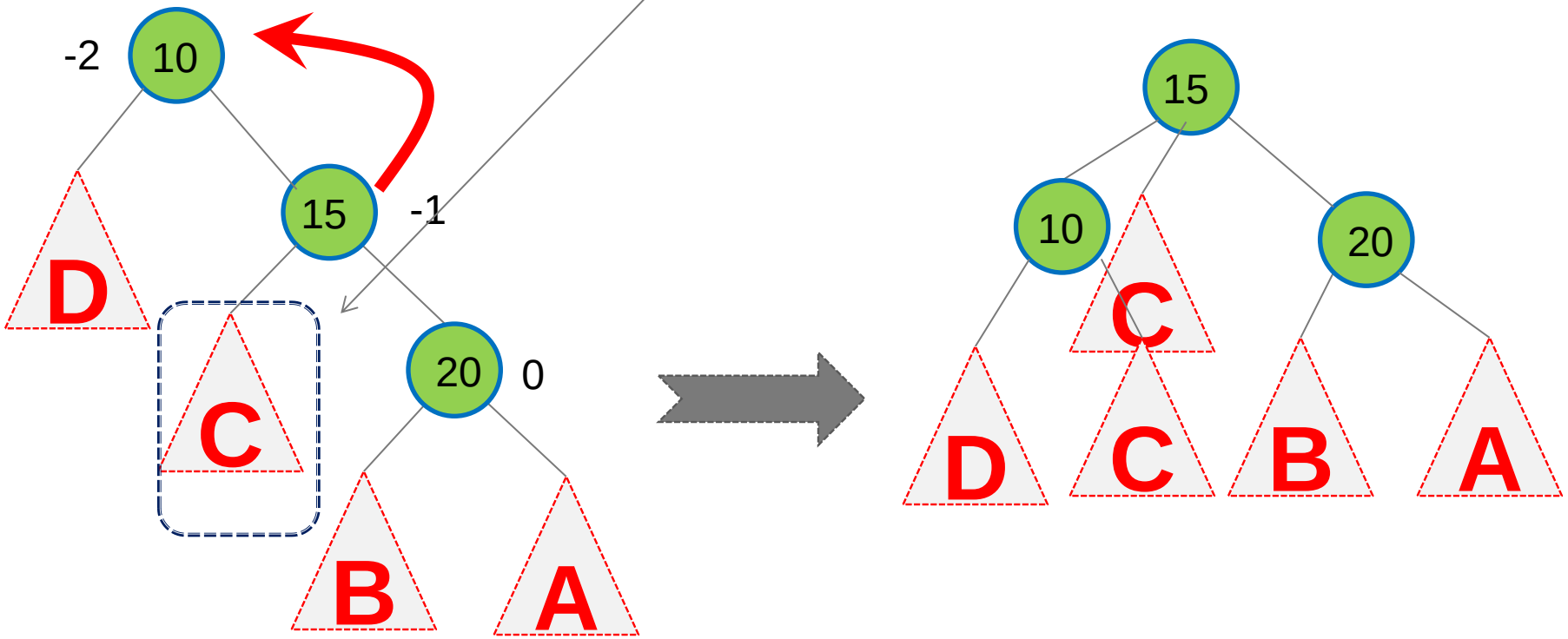
Note that all elements in B are greater than 20 and smaller than 24. Therefore, it can be made a left child of 24 after rotation.



Keeping AVL Tree Balanced

Right Right Case

Note that all elements in C are smaller than 15 and greater than 10. Therefore, it can be made a right child of 10 after rotation.

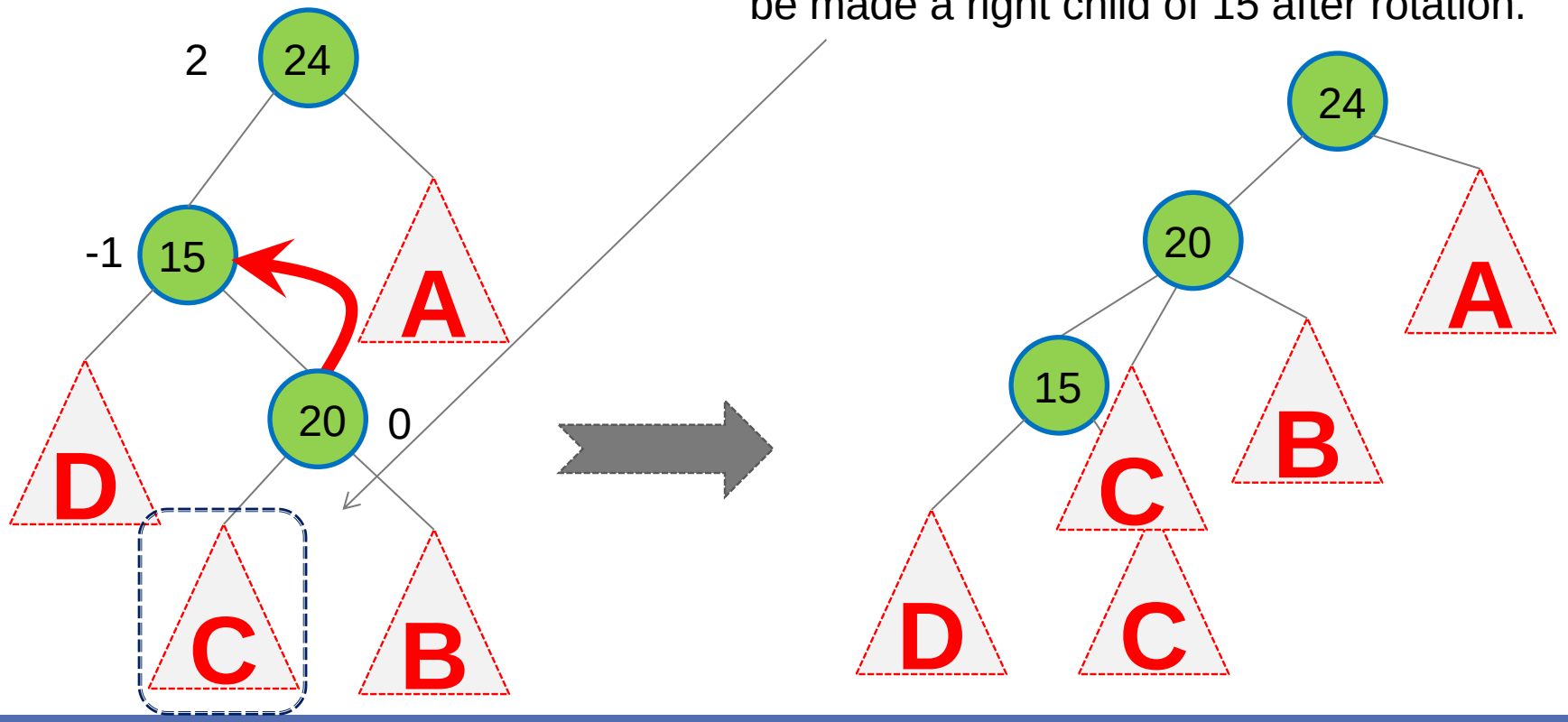


Keeping AVL Tree Balanced

Left Right Case

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

Note that all elements in C are smaller than 20 and greater than 15. Therefore, it can be made a right child of 15 after rotation.

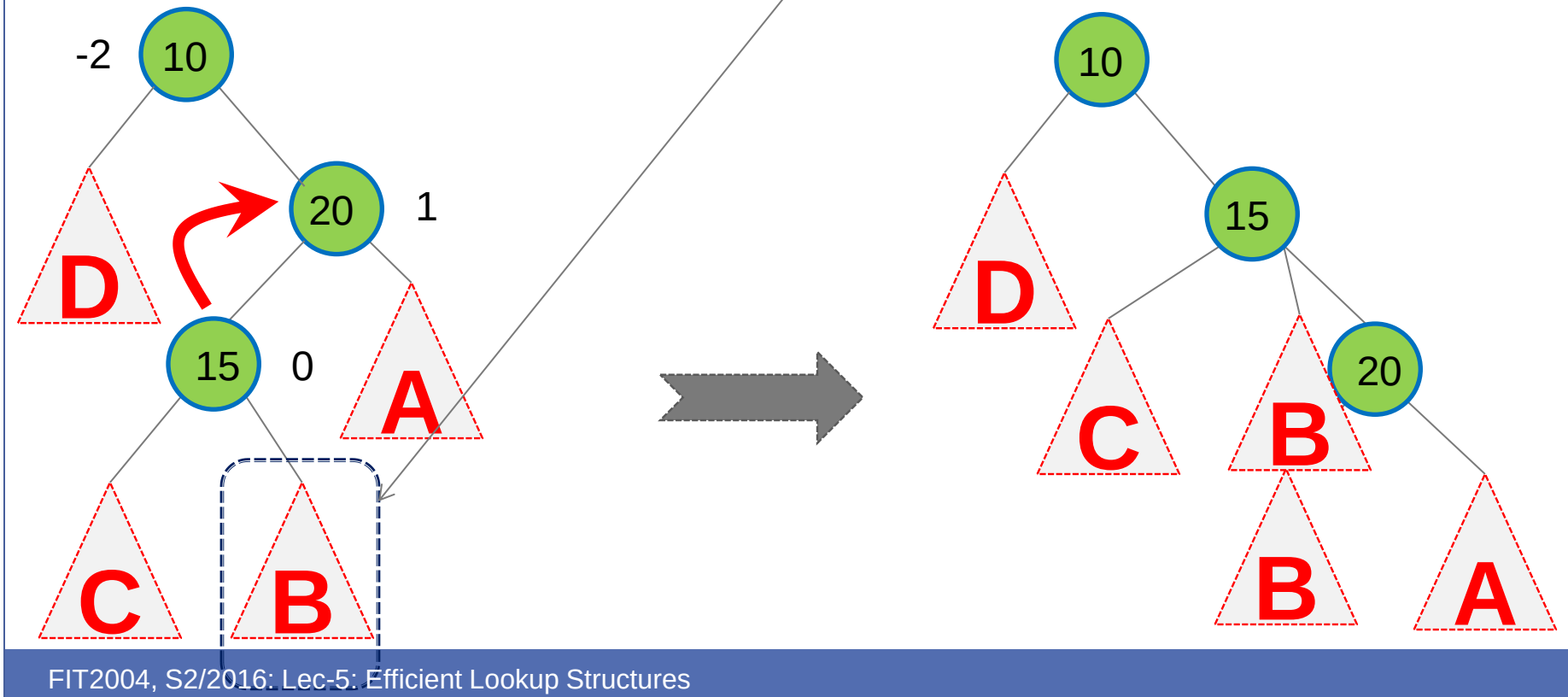


Keeping AVL Tree Balanced

Right Left Case

1. Convert Right Left Case to Right Right case by rotating 15
2. Handle Right Right case as earlier

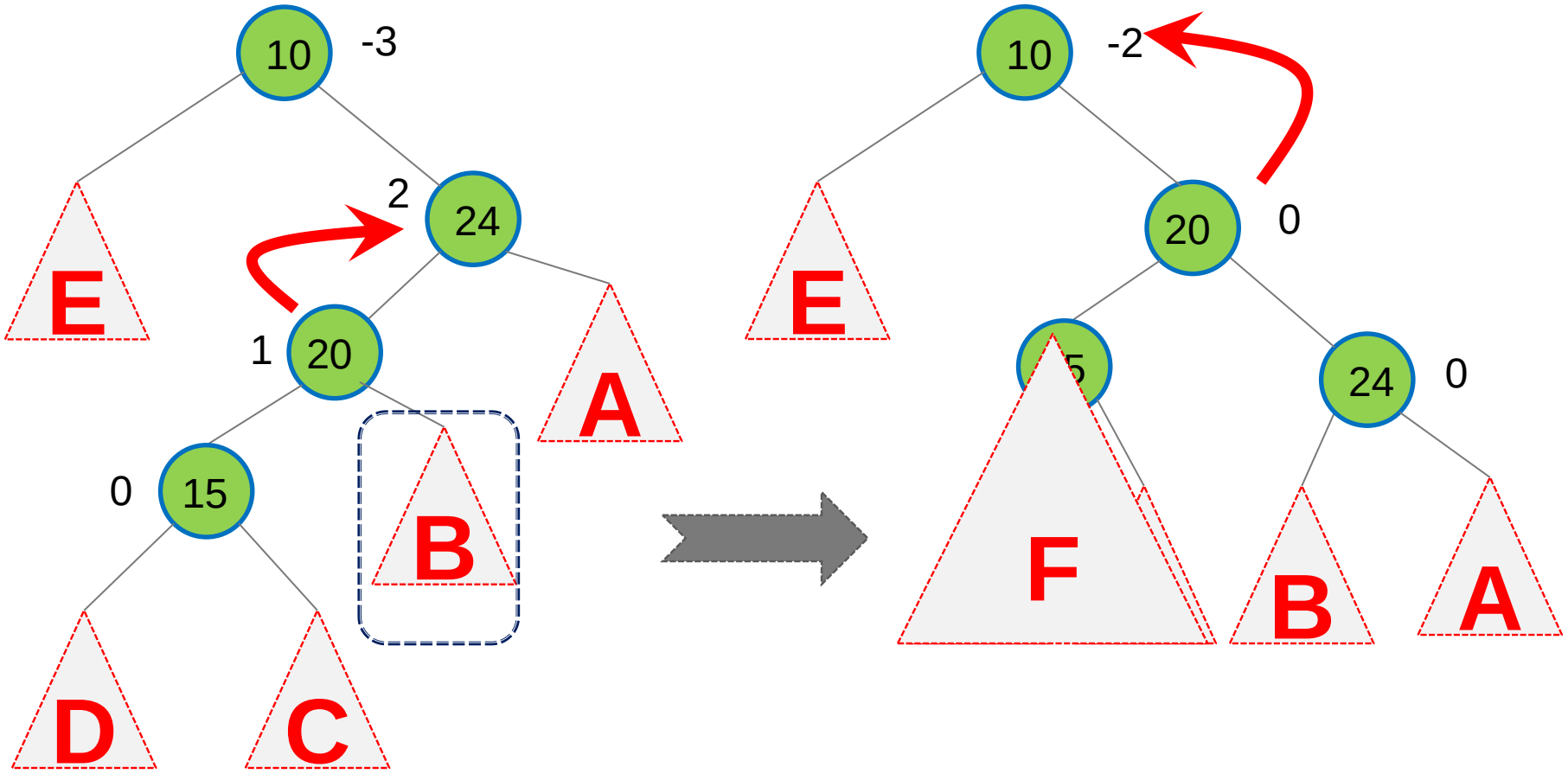
Note that all elements in B are greater than 15 and smaller than 20. Therefore, it can be made a left child of 20 after rotation.



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)
- Total nodes that require balancing are at most O(log N)



Searching and Deletion in AVL Tree

Searching in AVL Tree is exactly the same as in BST

- Worst-Case time complexity
  - O(log N) because the tree is balanced

Deletion in AVL Tree

- 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 time complexity: O(log N)

Summary

Take home message

- Hash tables provide O(1) look up in practice (although the worst-case time complexity is 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

- B-tree and Retrieval Trees