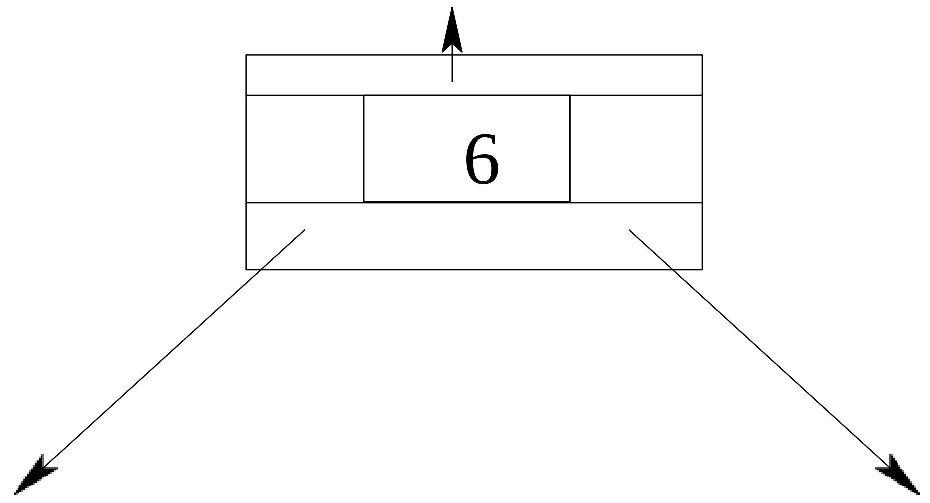
INT202
Complexity of Algorithms
Search Algorithms

XJTLU/SAT/INT

SEM2 AY2020-2021

Data structures for trees

- Linked structure: each node v of T is represented by an object with references to the element stored at v and positions of its parents and children.



Data structures for trees

- For rooted trees where each node has at most t children, and is of bounded depth, you can store the tree in an array A.

- Consider, for example, a binary tree. The root is stored in A[0]. The (possibly) two children of the root are stored in A[1] and A[2]. The two children of A[1] are stored in A[3] and A[4], and the two children of A[2] are in A[5] and A[6], and so forth.

Data structures for trees

- In general, the two children of node A[i] are in A[$2 * i + 1$] and
- A[$2 * i + 2$].
- The parent of node A[i] (except the root) is the node in position A[$\lfloor (i - 1)/2 \rfloor$].
- This can be generalized to the case when each node has at most t children

Ordered Data

- We often wish to store data that is ordered in some fashion (typically in numeric order, or alphabetical order, but there may be other ways of ordering it).
- Here we study a few methods for storing such ordered data, and maintaining the order as we add more data or remove it.
- In later lectures we will discuss efficient methods for *sorting* data into such an ordered collection. In these lecture notes we talk about methods for maintaining ordered data *as we store it* (which is different from sorting the data *after* we have already received the aggregated data as input).

Ordered Dictionary

In an ordered dictionary, we wish to perform the usual dictionary operation:

- ▷ *findElement(k)*: position *k*
- ▷ *insertElement(k,e)*: position *k* , element *e*
- ▷ *removeElement(k)*: position *k*

An *ordered* dictionary maintains an order relation for its elements, where the items are ordered by their keys.

Sorted Tables

If a dictionary *D*, is *ordered*, we can store its items in a vector, *S*, by *non-decreasing* order of keys. (This generally assumes that we're not going to add more items into the dictionary.)

Storing the dictionary in a vector in this fashion allows faster searching than if *S* is stored using a linked list.

A lookup table is a dictionary implemented by means of a sorted sequence

- We store the items of the dictionary in an array-based sequence, sorted by key
- We use an external comparator for the keys

Binary Search

▷ Accessing an element of *S* (in an array-based representation of size *n*) by its rank.

▷ The item at rank *i* has a key no smaller than keys of the items of ranks 1,...,*i* − 1 and no larger than keys of the items of ranks *i* + 1,*i* + 2,...,*n*.

The *search* is done on a decreasing range of the elements in *S*.

▷ Looking for *k* in *S*, the current range of *S* we consider is defined as a pair of ranks: *low* and *high*.

▷ Initially, *low* = 1 and *high* = *n*.

- ▷ *key*(*i*) denotes the key at rank *i*.
- ▷ *elem*(*i*) denotes the element at rank *i*.

Binary Search (cont.)

In order to *decrease* the size of the range we compare k to the key of the median, *mid*, of the range, i.e.,

$$mid = \lfloor (low + high)/2 \rfloor$$

Three cases are possible:

- ▷ $k = key(mid)$, the search is completed successfully.
- ▷ $k < key(mid)$, search continued with $high = mid - 1$.
- ▷ $k > key(mid)$, search continued with $low = mid + 1$.

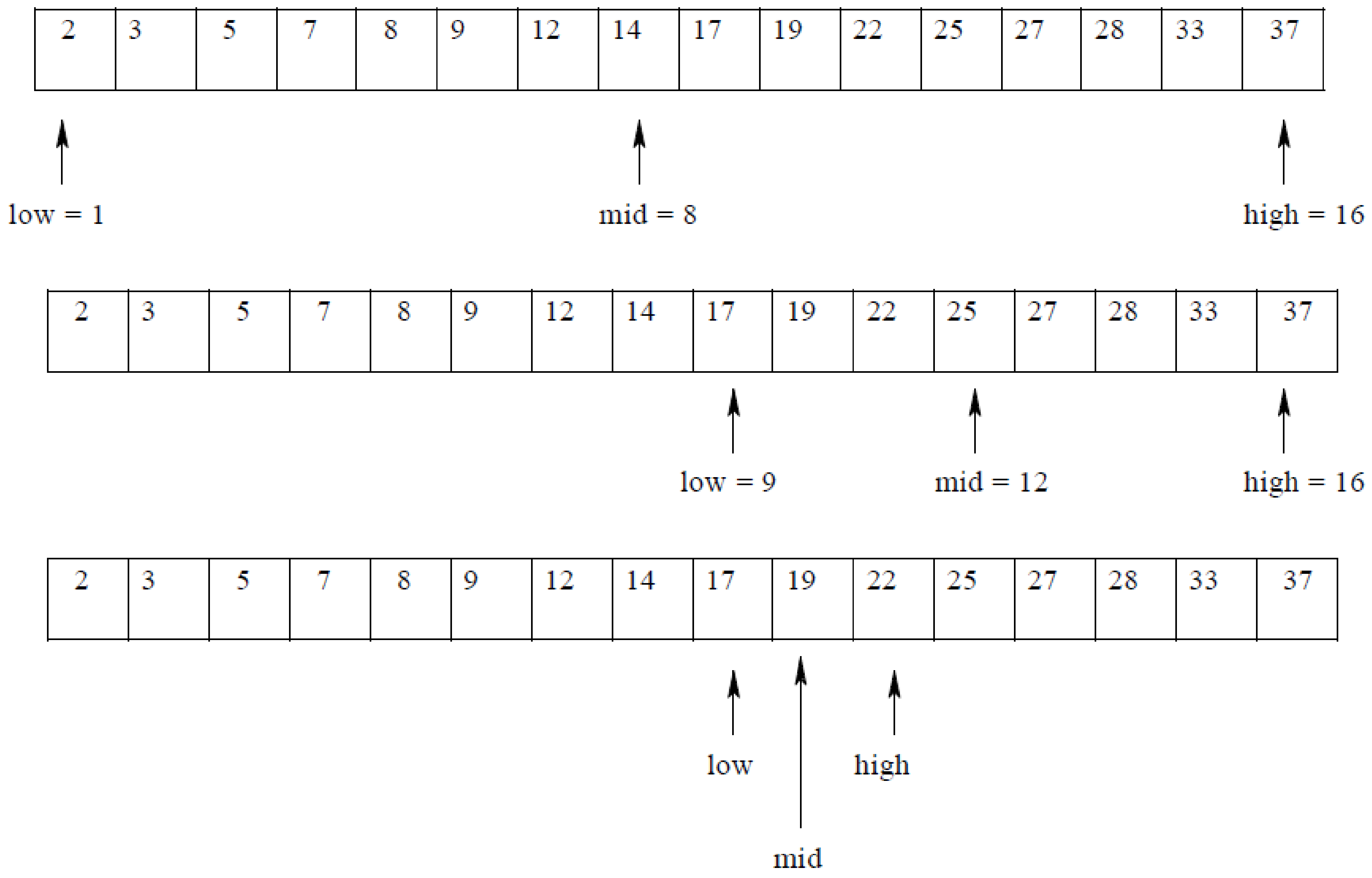
Binary Search - Algorithm

Here’s a recursive search algorithm.

```
BINARYSEARCH($S$, $k$,$low$,$high$)
  //Input is an ordered array of elements.
  //Output: Element with key $k$ if it exists, otherwise an error.
if $low > high$
    then return $NO\_SUCH\_KEY$                          // Not present
else
    $mid \leftarrow mid = \lfloor (low + high)/2 \rfloor$
      if $k = key(mid)$
          then return $elem(mid)$                         //Found it
      elseif $k < key(mid)$
          then return BINARYSEARCH($S$, $k$,$low$,$mid-1$)       // try bottom ‘half’
      else return BINARYSEARCH($S$,$k$,$mid + 1$,$high$)       //try top ‘half’
```

Binary Search - Example

- **B**INARY**S**EARCH(*S*,19,1,*size*(*S*))



Complexity of Binary Search

Considering the running time of binary search, we observe that a constant number of operations are executed at each recursive call.

Hence, the running time is proportional to the number of recursive calls performed.

The number of candidate items still to be searched in the array A is given by the value high− low +1.

Moreover, the number of remaining candidates is reduced by at least one half with each recursive call

$$(\text{mid} - 1) - \text{low} + 1 = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor - \text{low} \le \frac{\text{high} - \text{low} + 1}{2}$$

or

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor \le \frac{\text{high} - \text{low} + 1}{2}.$$

Complexity of Binary Search

Initially, the number of candidate is n; after the first call to BinarySearch, it is at most n/2; after the second call, it is at most n/4; and so on.

That is, if we let a function, T(n), represent the running time of this method, then we can characterize the running time of the recursive binary search algorithm as follows

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ T(n/2) + b & \text{otherwise} \end{cases}$$

where *b* is a constant that denotes the time for updating *low* and *high* and other overhead.

In general, this recurrence equation shows that the number of candidate items remaining after each recursive call is at most $n/2^i$.

The maximum number of recursive calls performed is the smallest integer such that $n/2^m < 1$. In other words, m > log n.

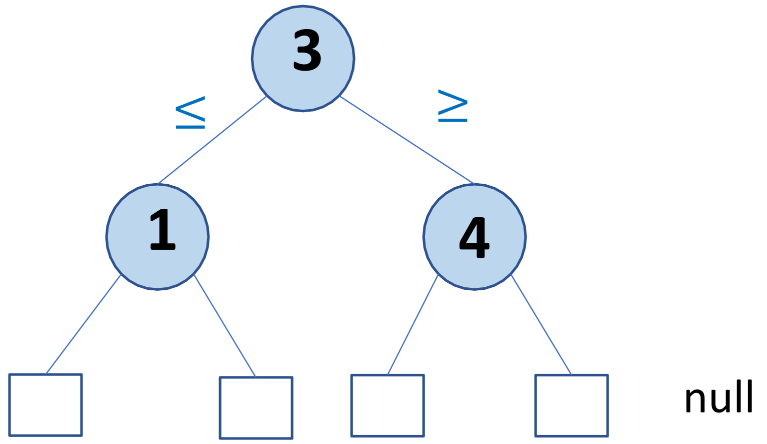
Thus, we have m = $\lfloor \log n \rfloor + 1$, which implies that BinarySearch(A, k, 1, n) runs in O(log n) time.

Binary Search Tree

A Binary Search Tree (*BST*) applies the motivation of binary search to a *tree-based* data structure.

In a *BST* each internal node stores an element, *e* (or, more generally, a key *k* which defines the ordering, and some element *e*).

A *BST* has the property that, given a node *v* storing the element *e*, all elements in the *left subtree* at *v* are *less than or equal to e*, and those in the *right subtree* at *v* are *greater than or equal to e*.



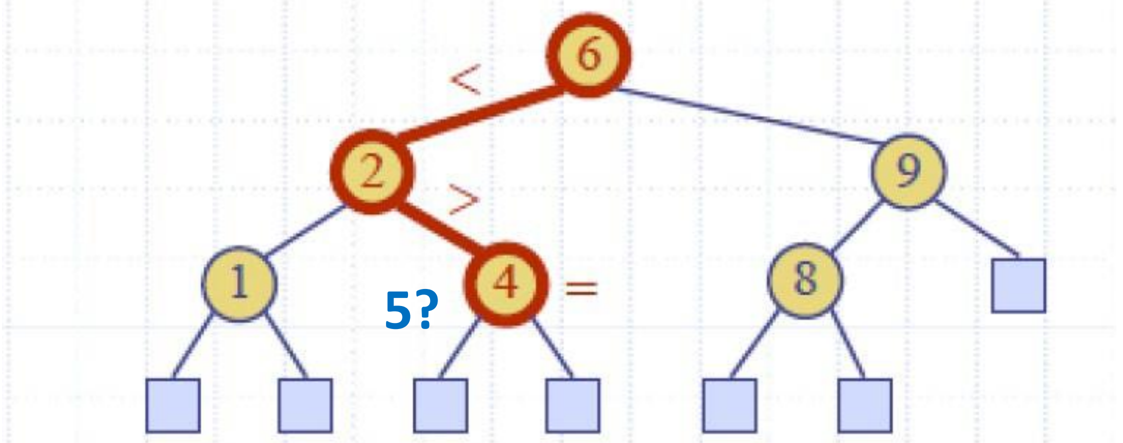
Inorder traversal of a BST: 1,3,4 (non-decreasing order)

Binary Search Tree

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return NO—SUCH KEY

Example: findElement(4)

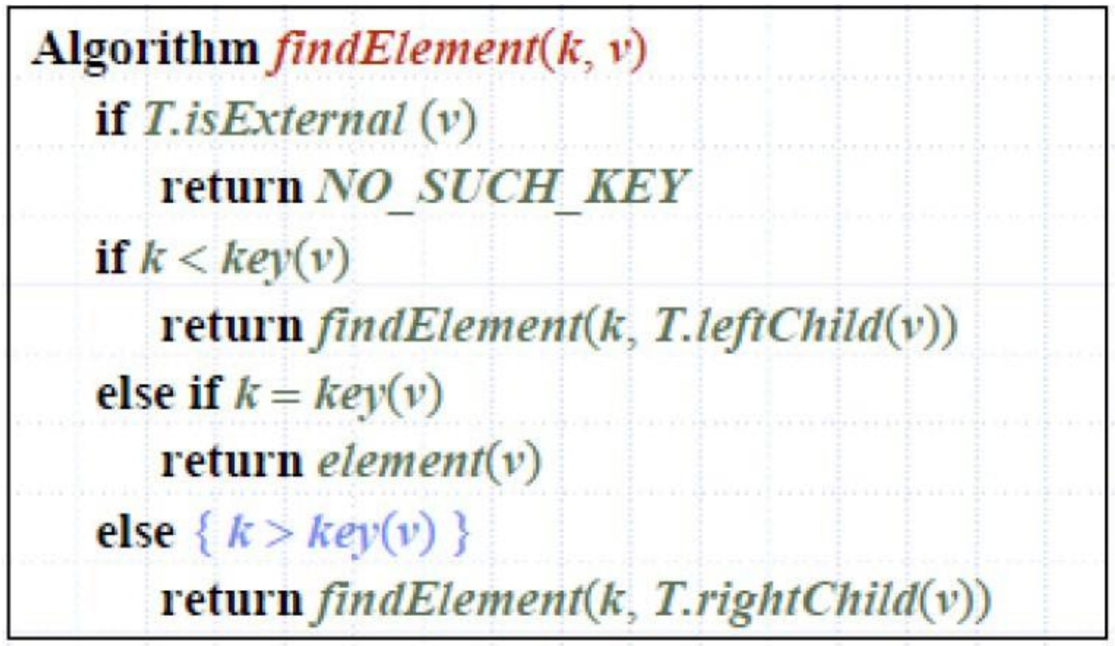
```
Algorithm findElement(k, v)
    if T.isExternal (v)
        return NO_SUCH_KEY
    if $k < key(v)$
        return findElement(k, T.leftChild(v))
    else if $k = key(v)$
        return element(v)
    else { $k > key(v)$ }
        return findElement(k, T.rightChild(v))
```



Binary Search Tree

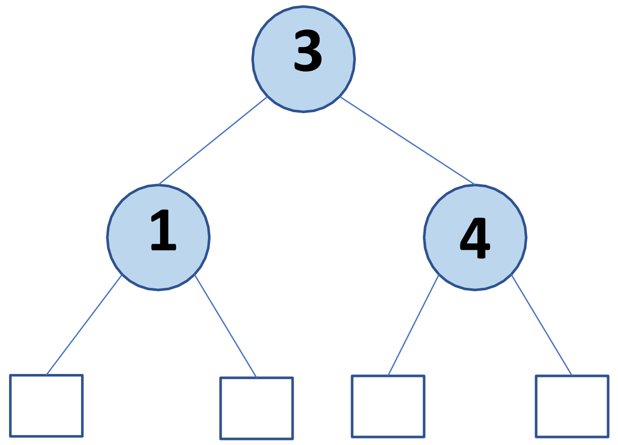
Time complexity

How long dose it takes in a n element tree?

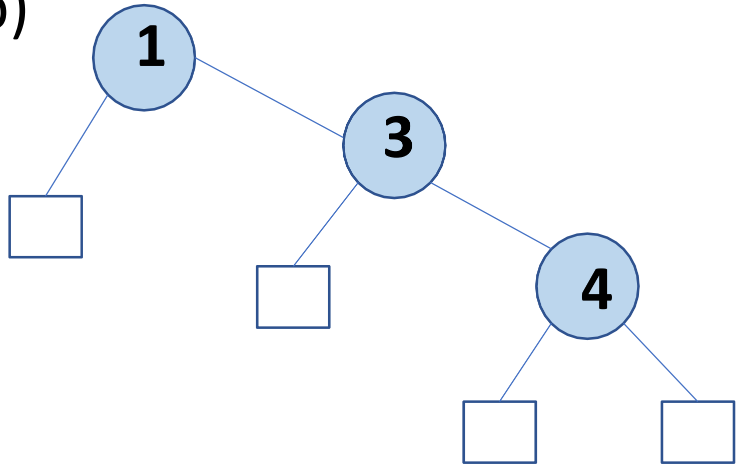


```
Algorithm findElement($k$, $v$)
    if T.isExternal ($v$)
        return NO_SUCH_KEY
    if $k < key(v)$
        return findElement($k$, T.leftChild($v$))
    else if $k = key(v)$
        return element($v$)
    else { $k > key(v)$ }
        return findElement($k$, T.rightChild($v$))
```

(a)

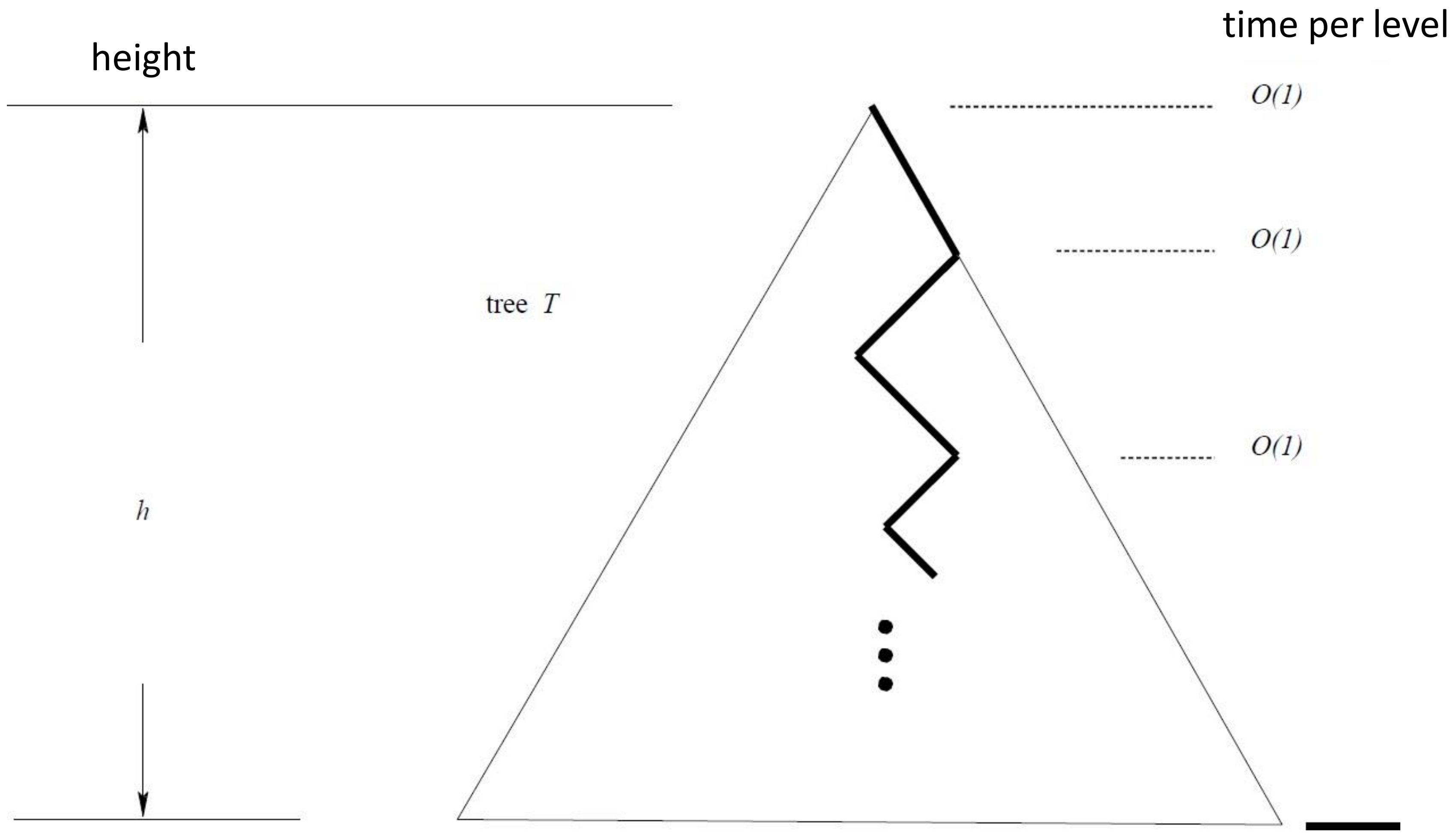


(b)



findElement(4)

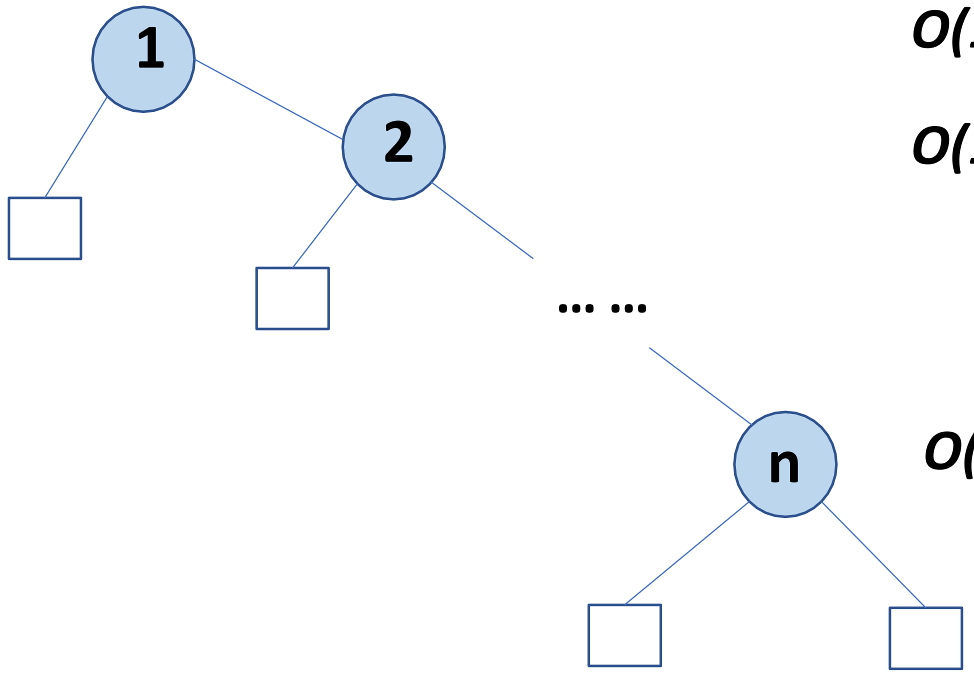
Complexity of searching in a BST



Total time O(h)

As small as log n or as large as n

findElement (n)



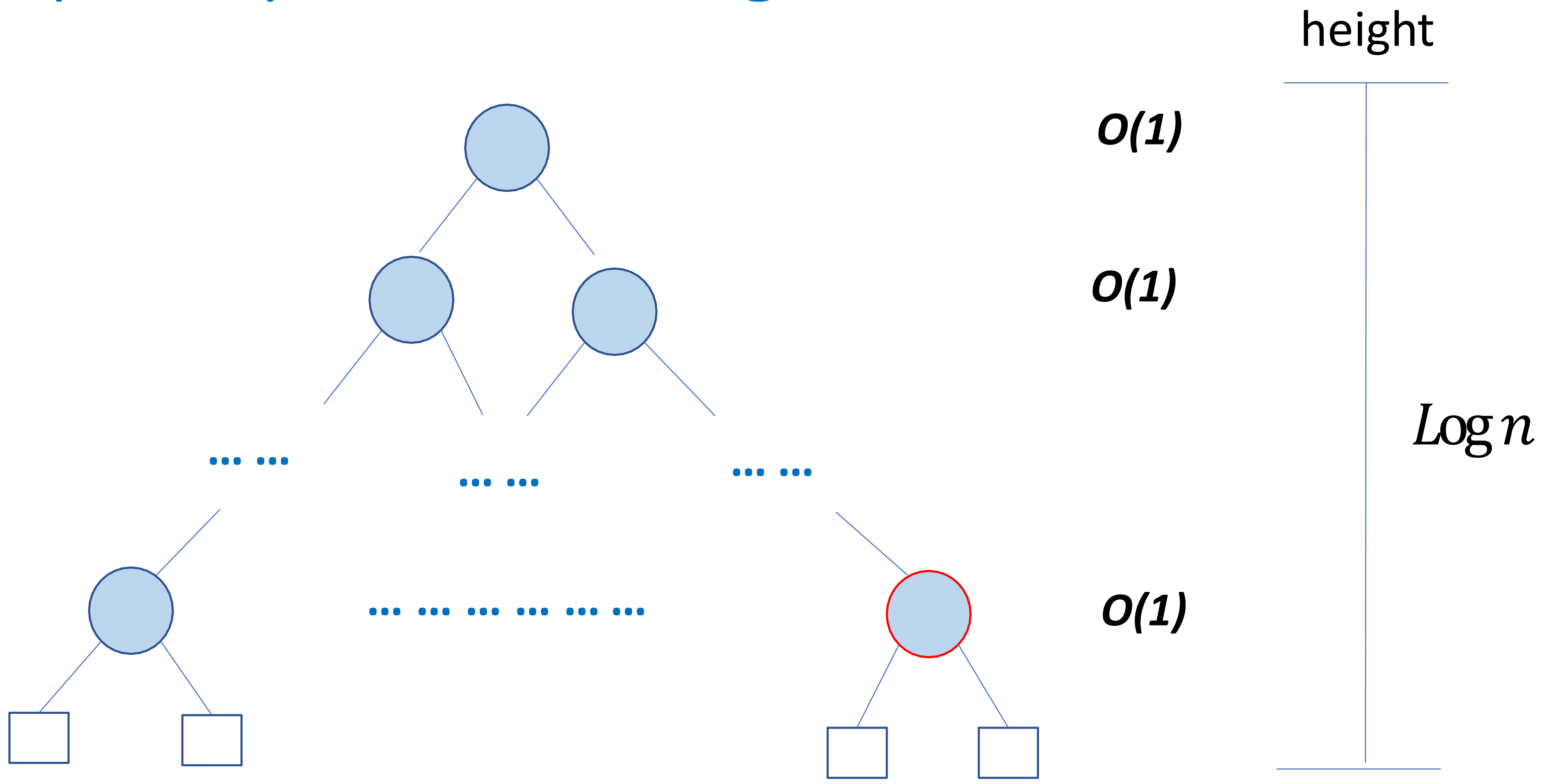
O(1)

O(1)

O(1)

Worse-case

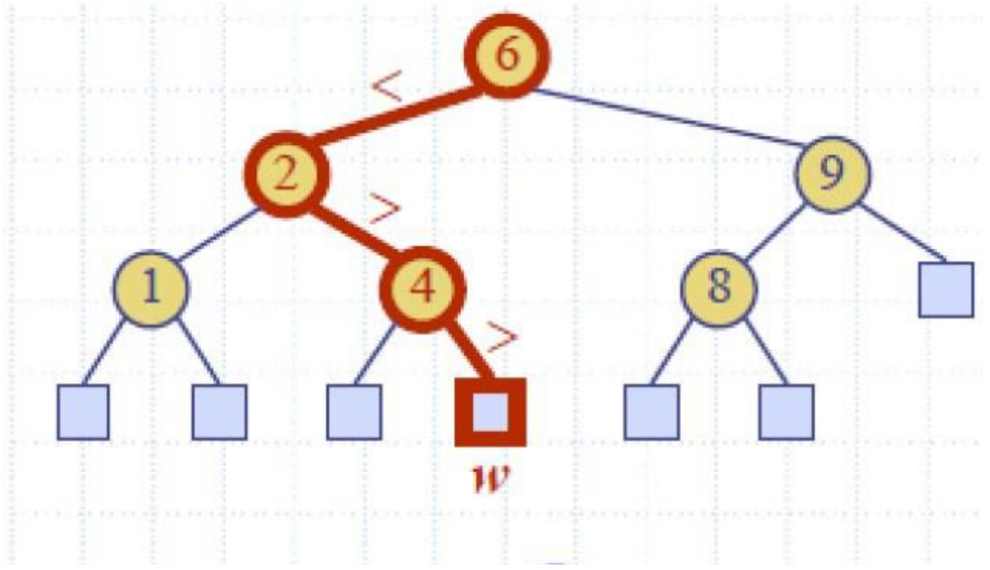
Complexity of searching in a BST



Insertion in a BST

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node

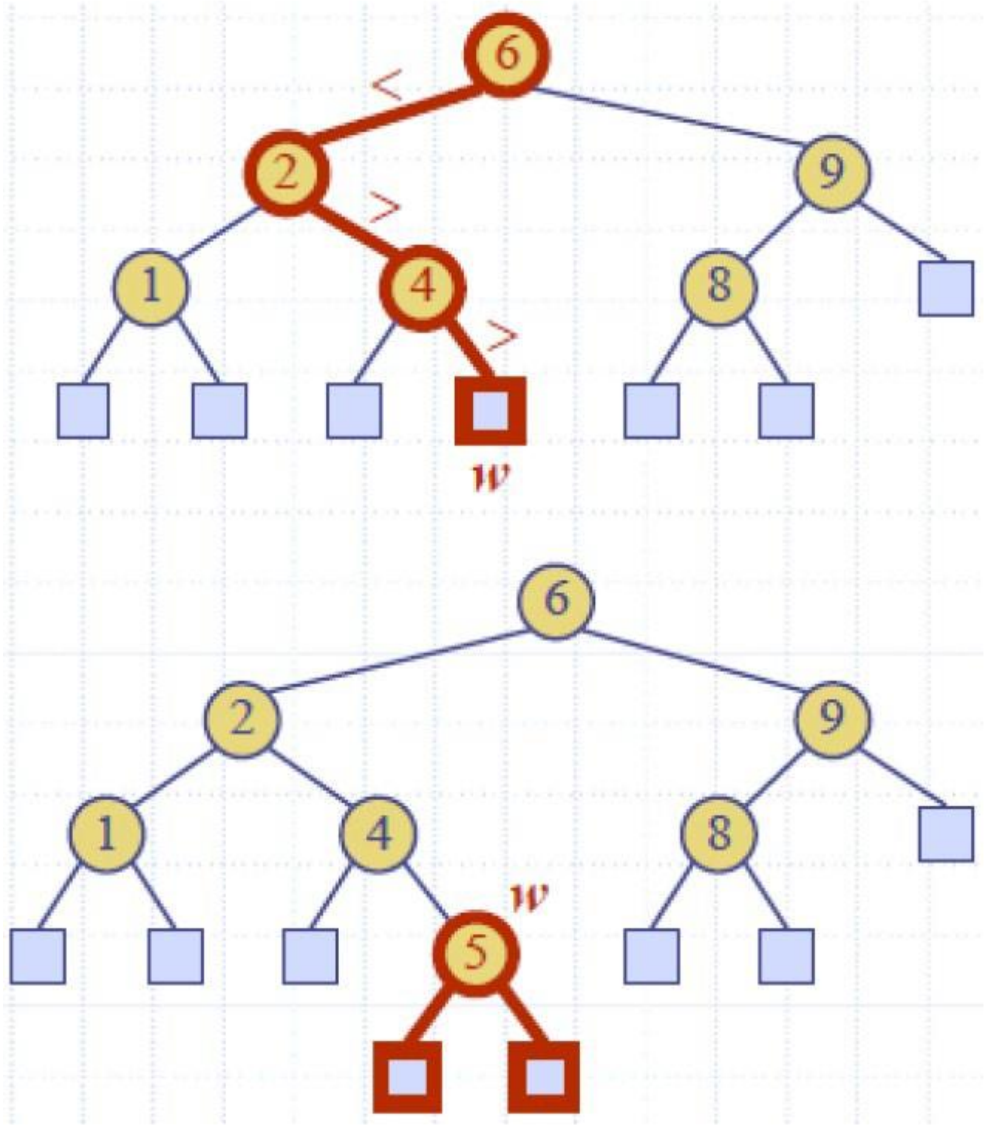
Example: insert 5



Insertion in a BST

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node

Example: insert 5



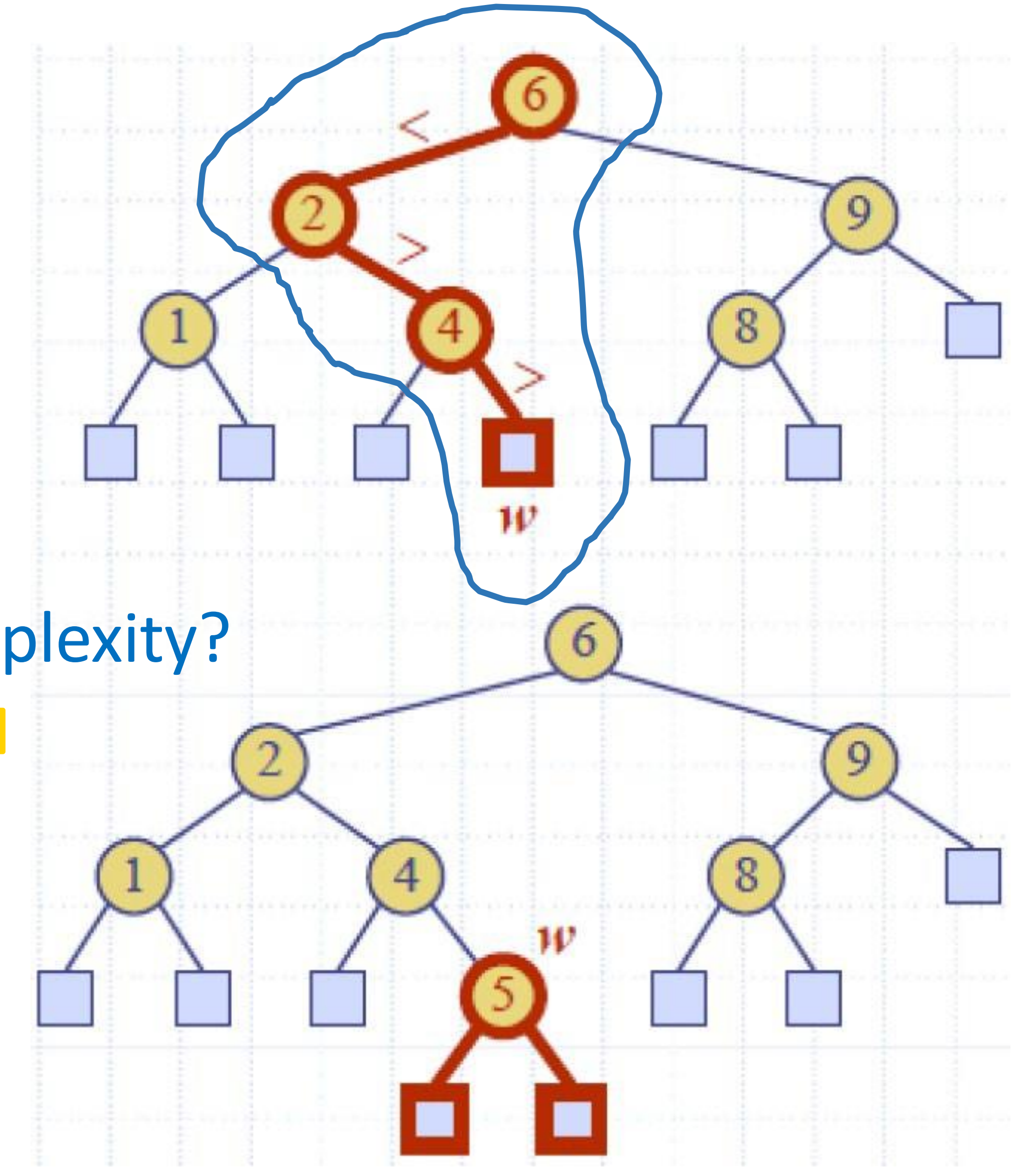
expandExternal(w)

Insertion in a BST

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node

Example: insert 5

Complexity?



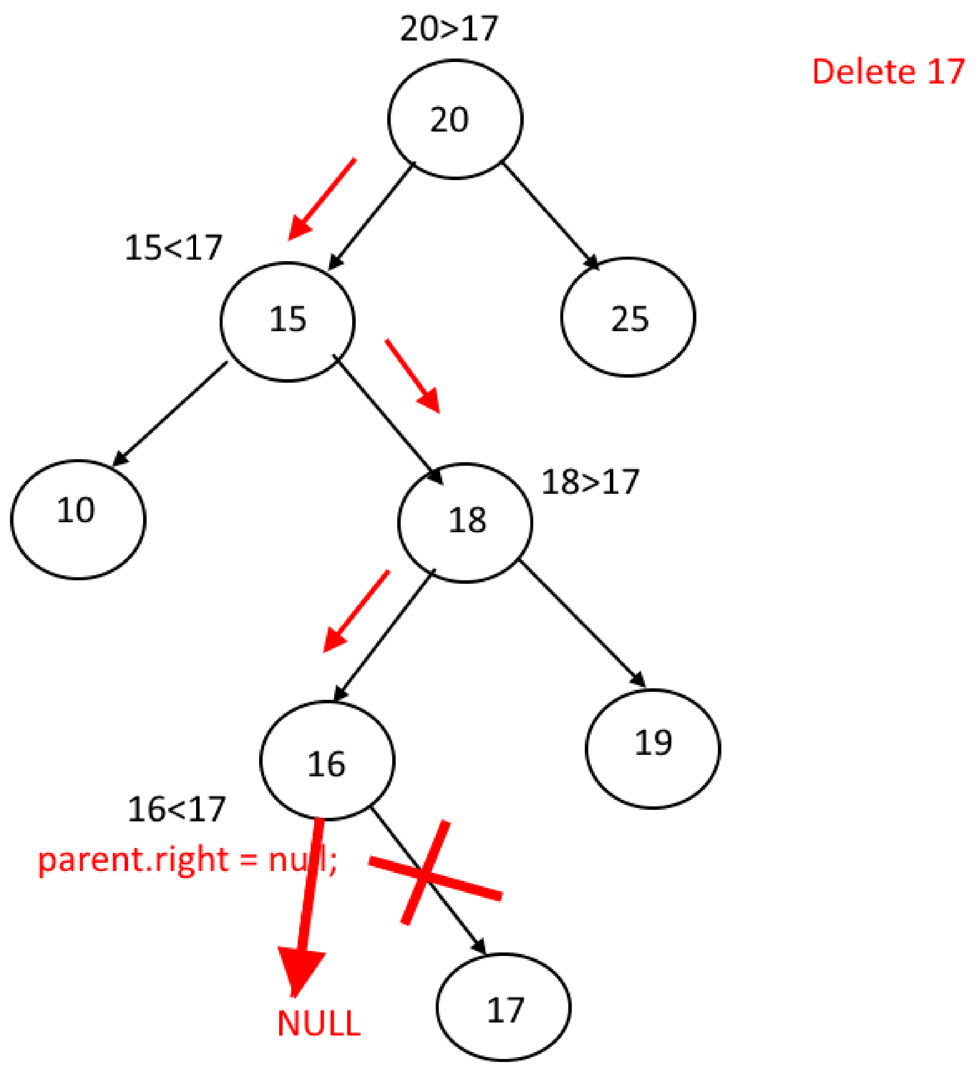
Deletion in a BST

- The following algorithm for deletion has the advantage of minimizing restructuring and unbalancing of the tree. The method returns the tree that results from the removal.

1. Find the node to be removed. We'll call it *remNode*.
2. If it's a leaf, remove it immediately by returning null.
3. If it has only one child, remove it by returning the other child node.
4. Otherwise, remove the *inorder successor* of *remNode*, copy its key into *remNode*, and return *remNode*.

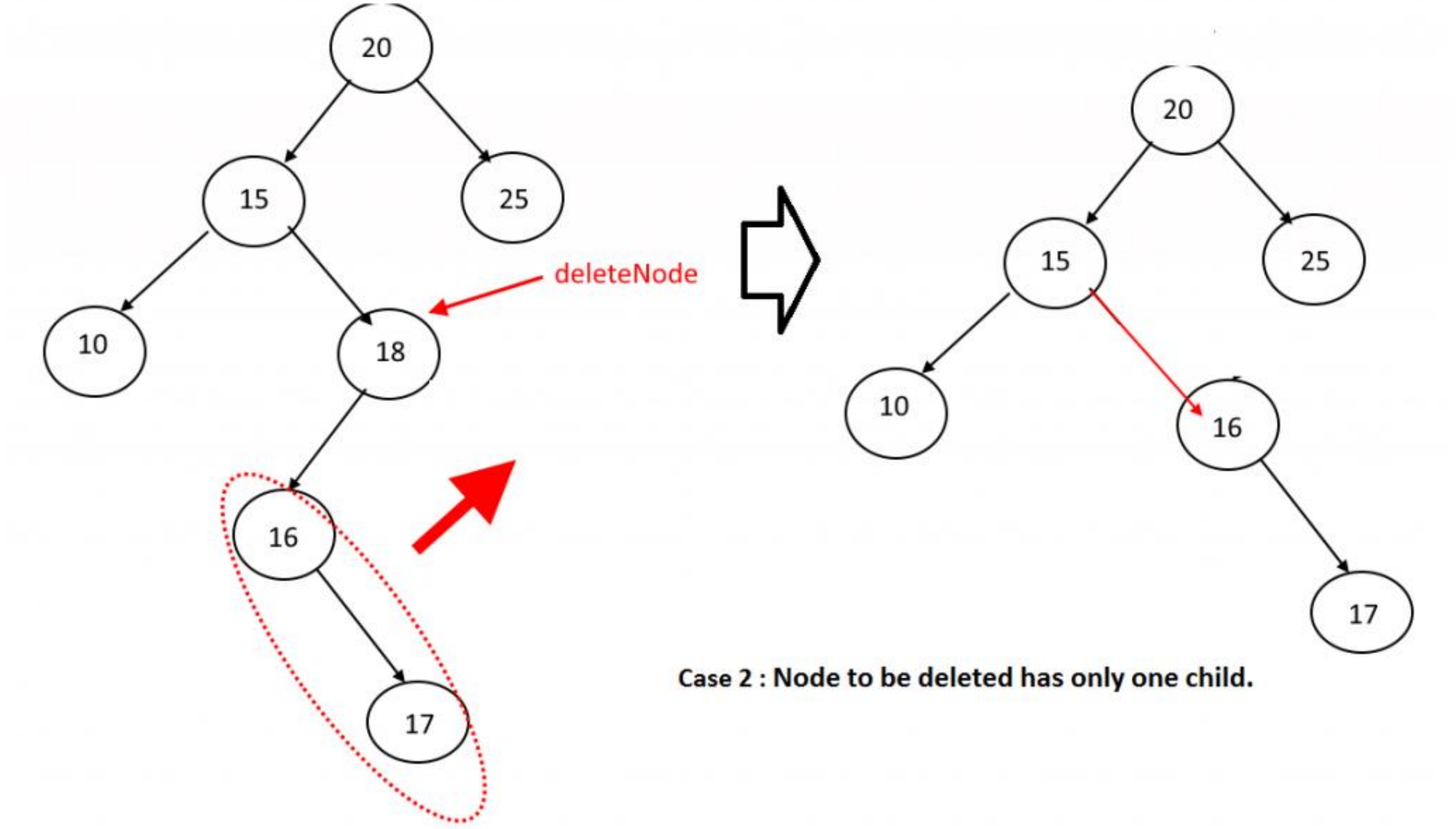
   (inorder successor: the node that would appear after the remNode if you were to do an inorder traversal of the tree)

Deletion in a BST



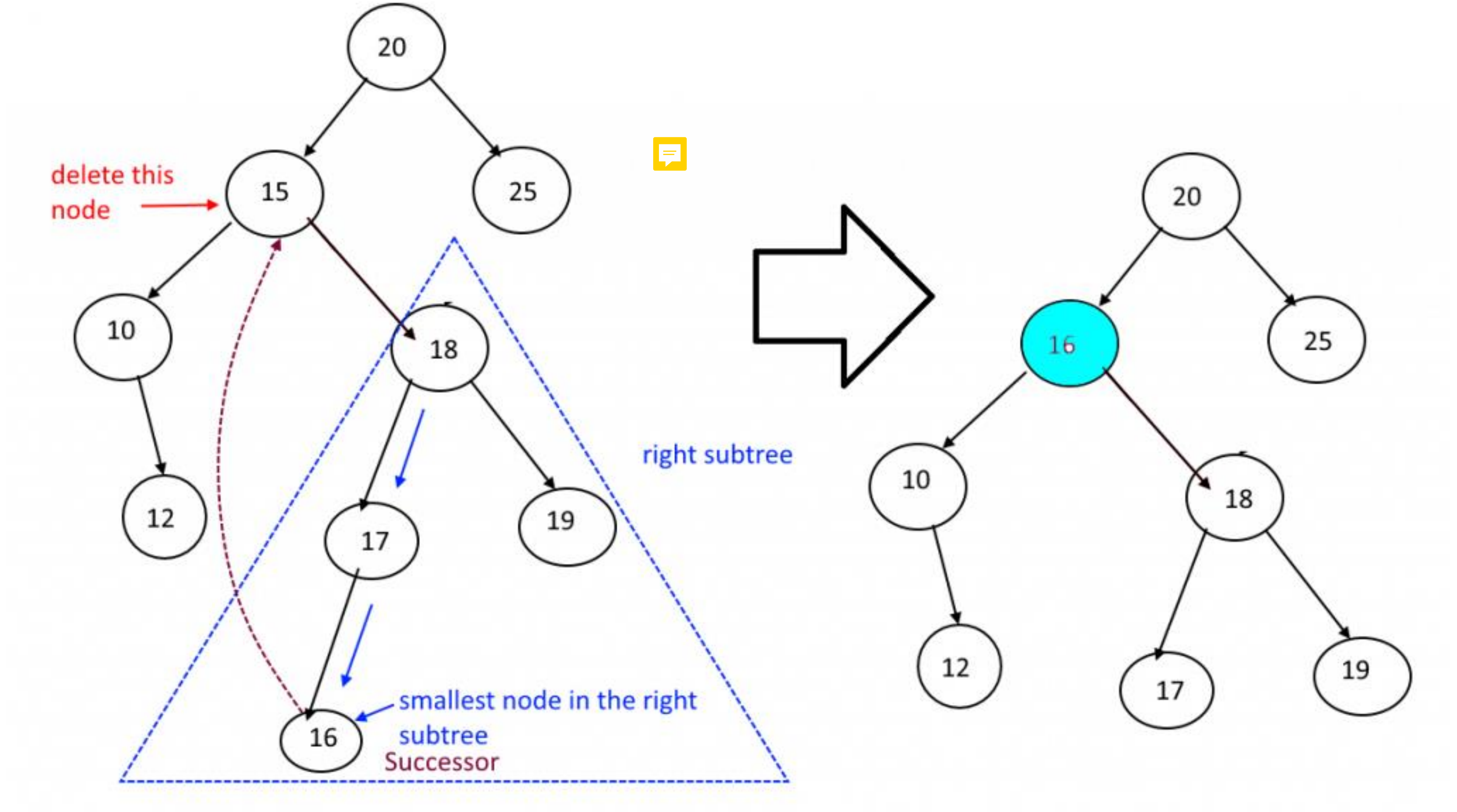
Case 1 : Node to be deleted is a leaf node (No Children).

Deletion in a BST



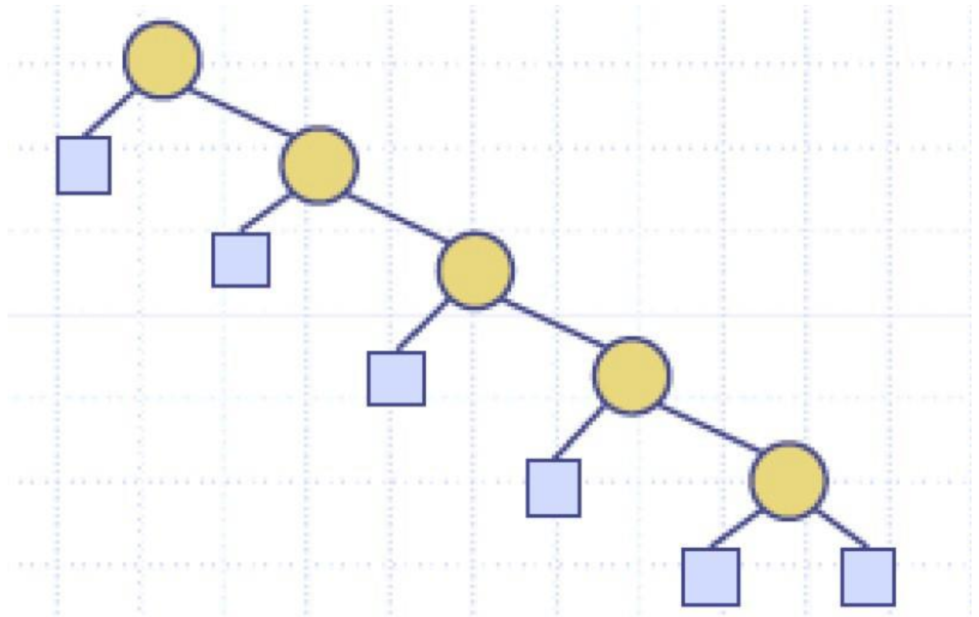
Case 2 : Node to be deleted has only one child.

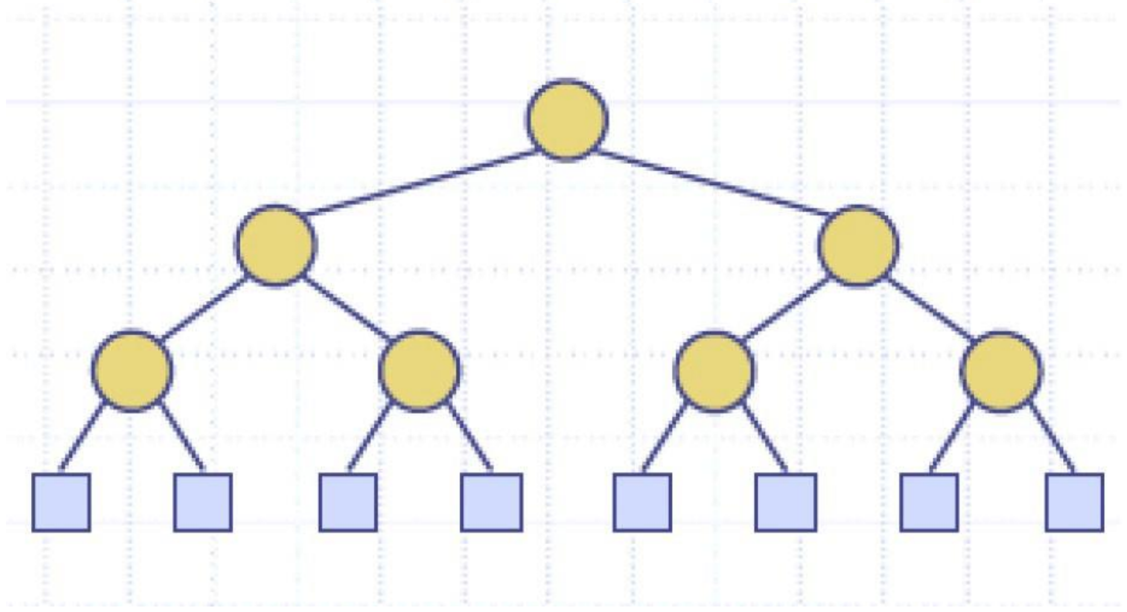
Deletion in a BST



BST Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is O(n)
  - method findElement take O(h) time
- The height h is O(n) in the worst case and O(log n) in the best case





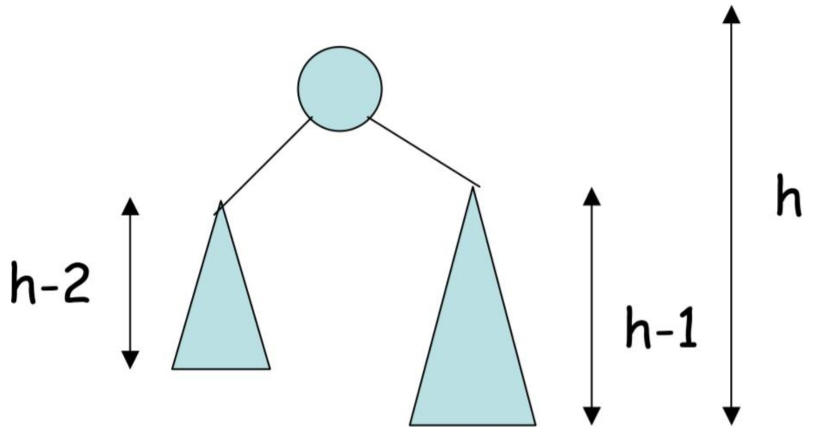
Inefficiency of general BSTs

▷ Unfortunately, *h* may be as large as *n*, e.g. for a *degenerate tree, where each parent node has only one associated child node (similar as linked list)*.

▷ The main advantage of binary searching (i.e. the $O(\log n)$ search time) may be lost if the BST is not *balanced*. In the worst case, the search time may be $O(n)$.

AVL trees (by Adel'son-Vel'skii and Landis)

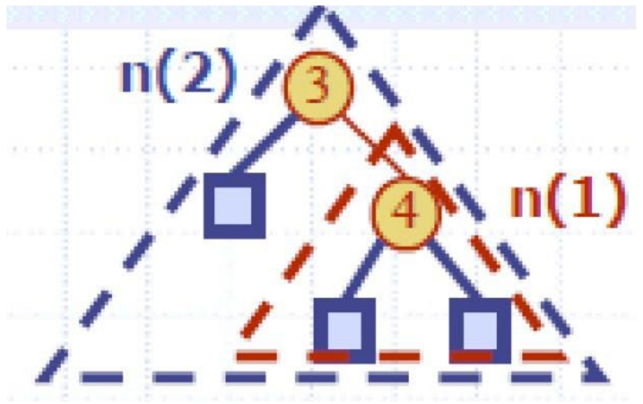
▷ *Height-Balance Property*: for any node n, the heights of n's left and right subtrees can differ by at most 1.



Height of an AVL Tree

- **Theorem**: The height of an AVL tree storing n keys is O(log n).

- **Theorem**: The height of an AVL tree storing n keys is O(log n).
- Proof: n(h): the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2.



- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
  n(h) > 2n(h-2), n(h-2) > 2n(h-4), n(h-4) > 2n(n-6) => n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$ $\rightarrow$ **h-2i = 1 or 2, depending on if h is even or odd: i = ⌈h/2⌉-1.**
  Taking logarithms: h < 2log n(h) +2 $\rightarrow$ **log n(h) > log $2^{h/2-1}$, log n(h)>h/2-1**
- Thus the height of an AVL tree is O(log n)

AVL trees (by Adel'son-Vel'skii and Landis)

An *AVL tree* is a tree that has the Height-Balance Property.

- **Theorem**: The height of an *AVL tree* storing n keys is $O(\log n)$.

Consequence 1: A *search* in an AVL tree can be performed in time $O(\log n)$.

Consequence 2: Insertions and removals in AVL need more careful implementations (using *rotations* to maintain the height-balance property).

Insertion & deletion in AVL trees

An insertion in an AVL tree begins as an insertion in a general BST, i.e., attaching a new external node (leaf) to the tree.

- ►This action may result in a tree that *violates* the *height-balance property* because the heights of some nodes increase by 1.

A deletion in an AVL tree begins as a removal in a general BST.

- ►Action may violate the height-balance property.