COMP20010: Algorithms and Imperative Programming

Lecture 4

Ordered Dictionaries and Binary Search Trees

AVL Trees

Lecture outline

- Sorted tables;
- Binary search trees
- Searching, inserting and removal with binary search trees;
- Performance of binary search trees;
- AVL trees (update operations, performance);

Ordered dictionaries

- In an ordered dictionary we use a comparator to provide the order relation among the keys. Such ordering allows efficient implementation of the dictionary ADT.
- An ordered dictionary supports the following methods:
  - closestKeyBefore(k) – returns the largest key that is $\leq k$;
  - closestElemBefore(k) – returns e with the largest key that is $\leq k$;
  - closestKeyAfter(k) – returns the smallest key that is $\geq k$;
  - closestElemAfter(k) – returns e with the smallest key that is $\geq k$;
- The ordered nature of the above operations makes the use of a log file or a hash table inappropriate for implementing the dictionary (neither of the two data structures has any ordering among the keys).

Sorted Tables

- If a directory D is ordered, the items can be stored in a vector S by non-decreasing order of the keys.
- The ordering of the keys allows faster searching than in the case of un-ordered sequences (possibly implemented as a linked list).
- The ordered vector implementation of a dictionary D is referred to as the **lookup table**.
- The implementation of insertItem(k,e) in a lookup table takes $O(n)$ time in the worst case, as we need to shift up all the items with keys greater than k to make room for the new item.
- On the other hand, the operation findElement is much faster in a sorted lookup table than in a log file.

Binary Search

- If *S* is an ordered sequence, than the element at the rank (position) *i* has a key that is not smaller than the keys of the items at ranks $0,\ldots,i-1$ and no larger than the keys of the items at ranks $i+1,\ldots.n-1$.
- This ordering allows quick searching of the sequence *S* using a variant of the game "high-low". The algorithm has two parameters: **high** and **low**. All the candidates for a sought element at a current stage of the search are bracketed between these two parameters, i.e. they lie in the interval [low,high].
- The algorithm starts with the values low=0 and high=*n*-1. Then the key *k* of the element we are searching for is compared to a key of the element at a half of *S*, i.e. $\text{mid} = \lfloor(\text{low}+\text{high})/2\rfloor$. Depending on the outcome of this comparison we have 3 possibilities:
  - If *k*=key(mid), the item we are searching for is found and the algorithm terminates returning *e*(mid);
  - If *k*<key(mid), then the element we are searching for is in the lower half of the vector *S*, and we set high=mid-1 and call the algorithm recursively;
  - If *k*>key(mid), the element we are searching for is in the upper half of the vector *S*, and we set low=mid+1 and call the algorithm recursively;

Binary Search

- Operation findElement(*k*) on an *n*-item ordered dictionary implemented with a vector *S* reduces to calling BinarySearch(*S,k,0,n*-1).

  Algorithm BinarySearch(*S,k,*low,high):

  Input: An ordered vector *S* storing *n* items, a search key *k*, and the integers low and high;

  Output: An element of *S* with the key *k*, otherwise an exception;

  if low>high **then**

  return NO_SUCH_KEY

  else

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

  if *k*=key(mid) **then**

  return *e*(mid)

  elseif *k*<key(mid) **then**

  BinarySearch(*S,k,*low,mid-1)

  else

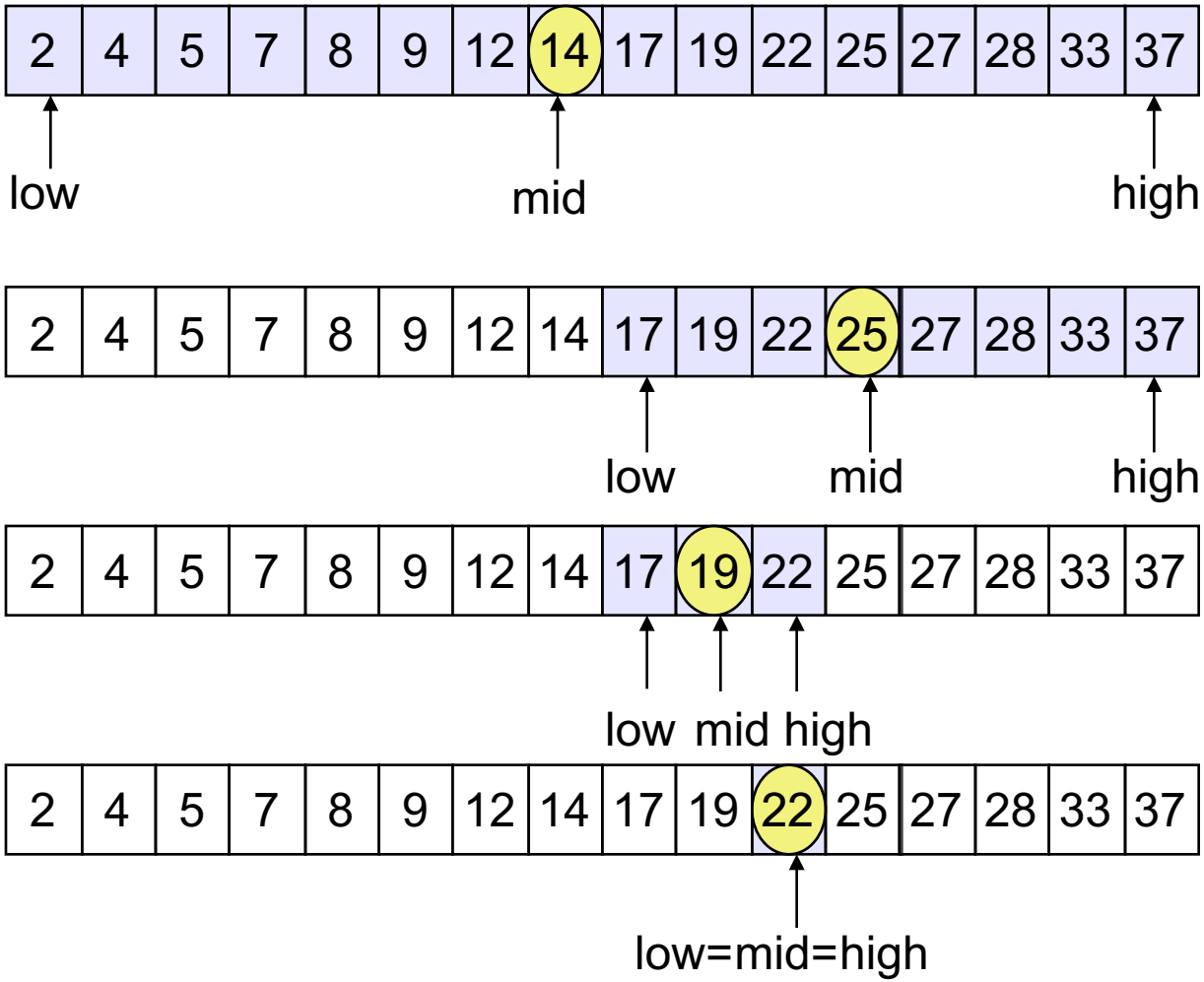
  BinarySearch(*S,k,*mid+1,high)

  endif

  end if

Binary Search

An example of binary search to perform the operation findElement(22)



Binary Search

- Considering the computational cost of binary search, we need to notice first that at each call of the algorithm there is a constant number of operations. Thus the running time is proportional to the number of recursive calls.
- At each recursive call the number of candidates that need to be searched is high-low+1, and it is reduced by at least a half at each recursive call.
- If $T(n)$ is the computational cost of binary search, then:

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

- In the worst case the search stops when there are no more candidate items. Thus, the maximal number of recursive calls is m, such that

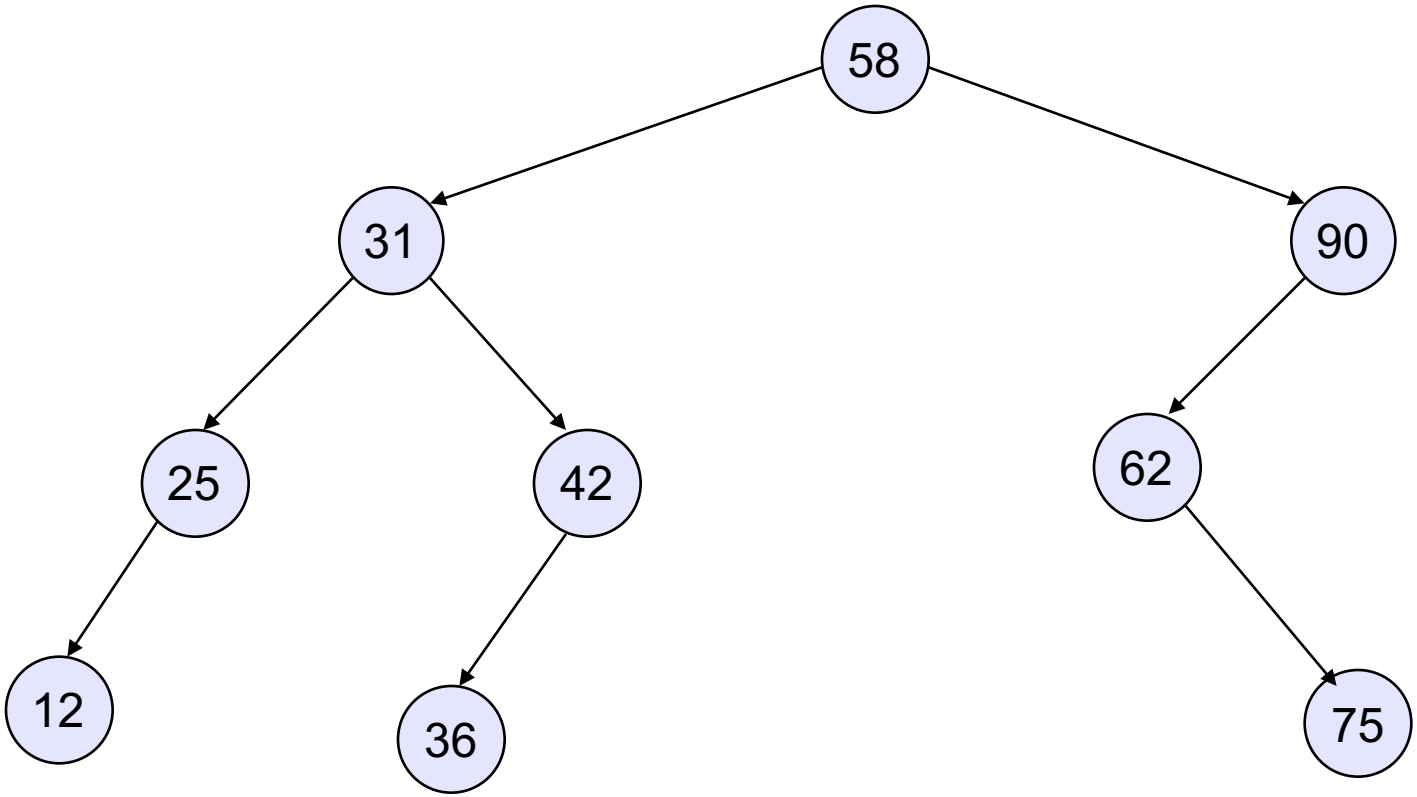
$$n/2^m < 1$$

- This implies that $m = \lfloor \log n \rfloor + 1$, i.e. BinarySearch($S$,$k$,0,$n$-1) runs in $O(\log n)$ time.

Binary Search Trees

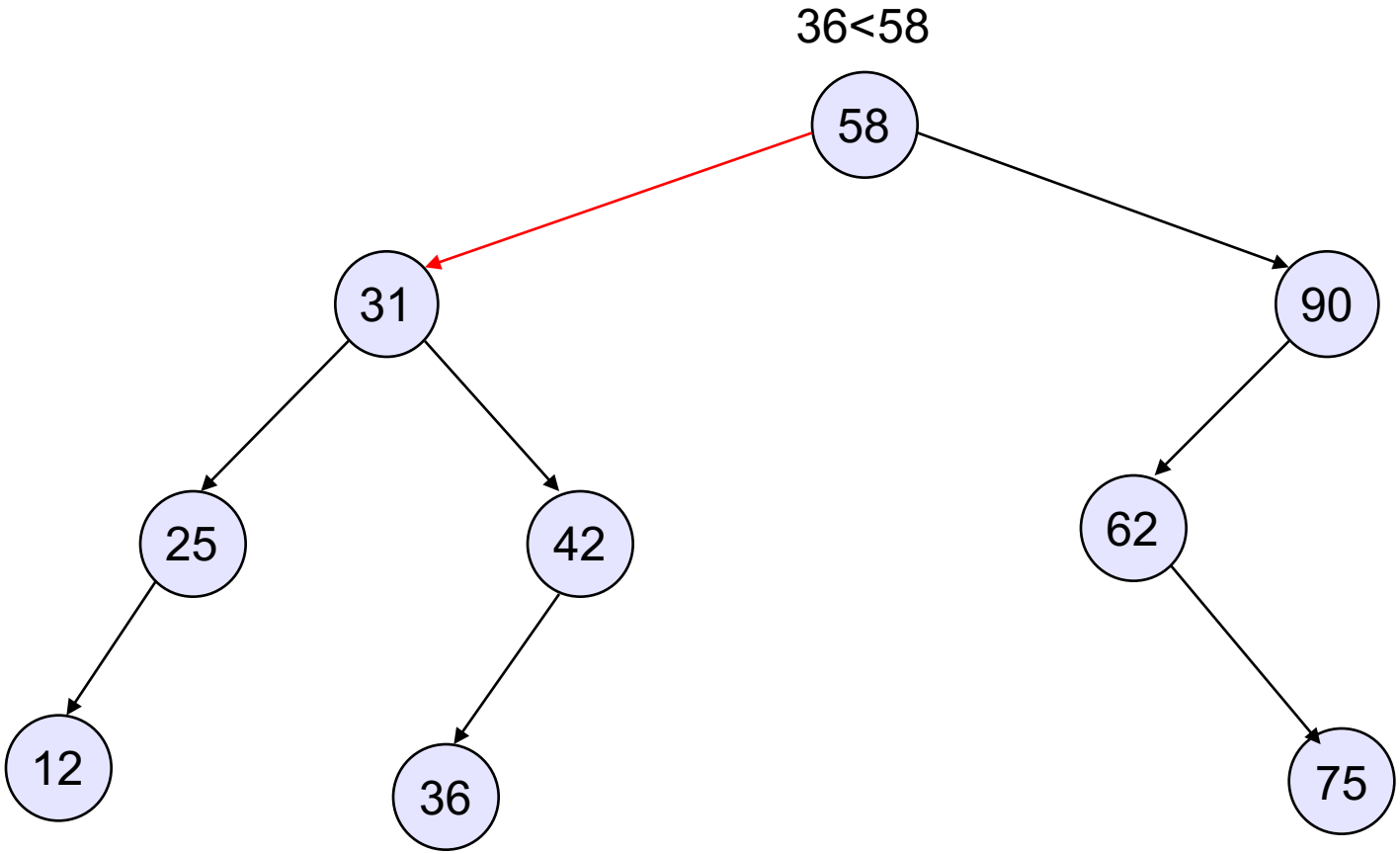
- It is a tree data structure adapted to a binary search algorithm.
- A binary search tree is a binary tree in which each node stores an element *e* and that the elements in the left subtree of that node are smaller or equal to *e*, while the elements in the right subtree of that node are greater or equal to *e*.
- An inorder traversal of a binary search tree visits the elements in a non-decreasing order.
- A binary search tree can be used to search for an element by traversing down the tree. At each node we compare the value we are searching for *x* with *e*. There are 3 outcomes:
  - If $x=e$, the search terminates successfully;
  - If $x<e$, the search continues in the left subtree;
  - If $x>e$, the search continues in the right subtree;
  - If the whole subtree is visited and the element is not found, the search terminates unsuccessfully;

Binary Search Trees



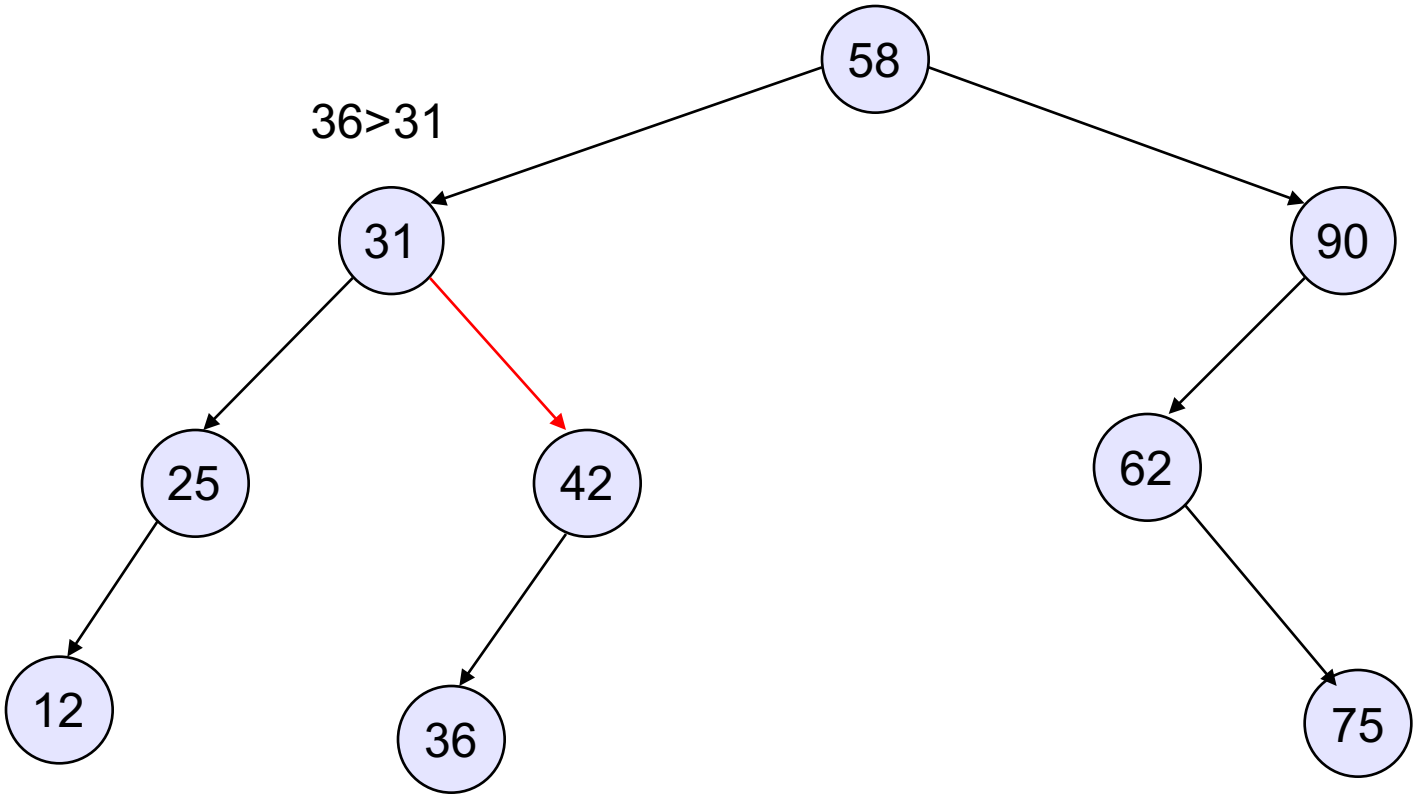
Searching for the element 36

Binary Search Trees



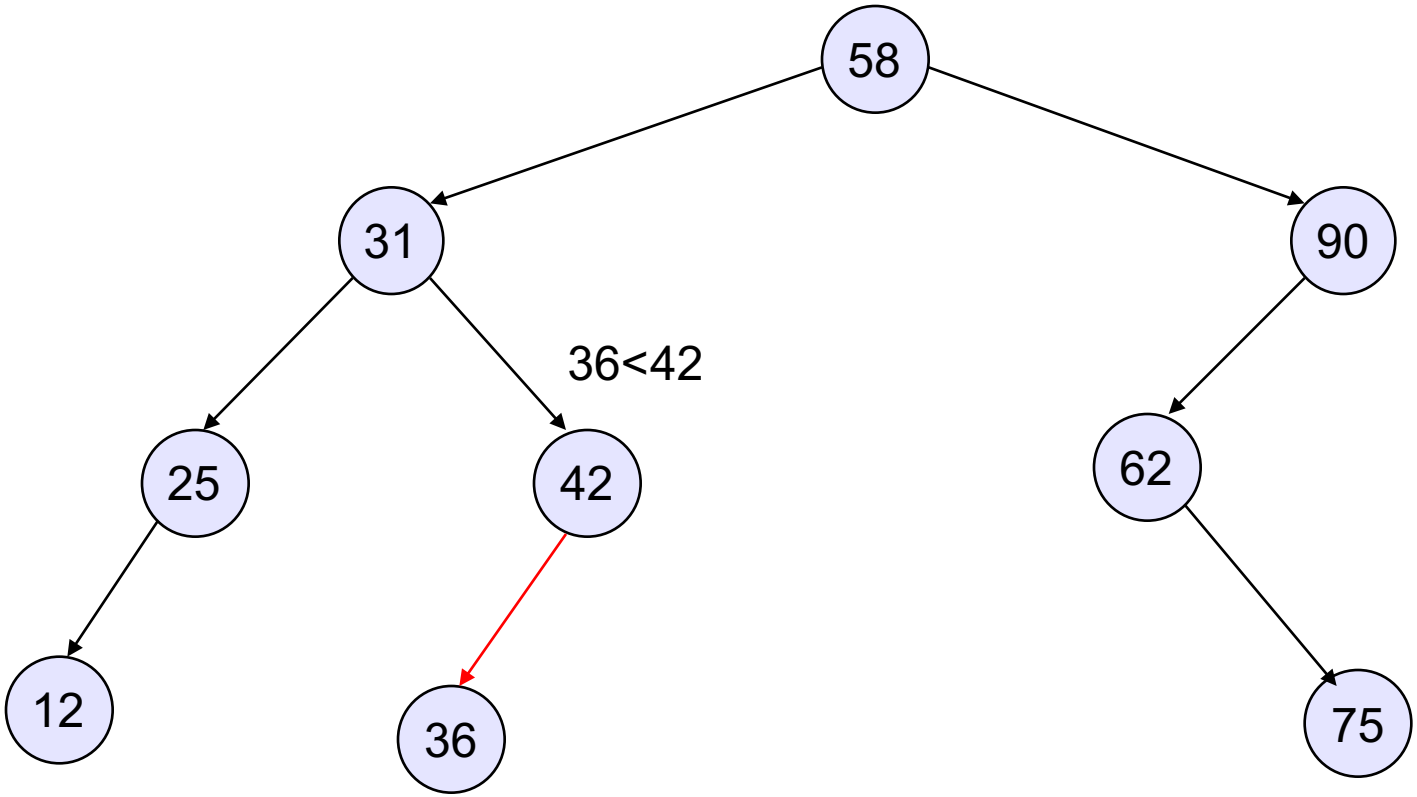
Searching for the element 36

Binary Search Trees



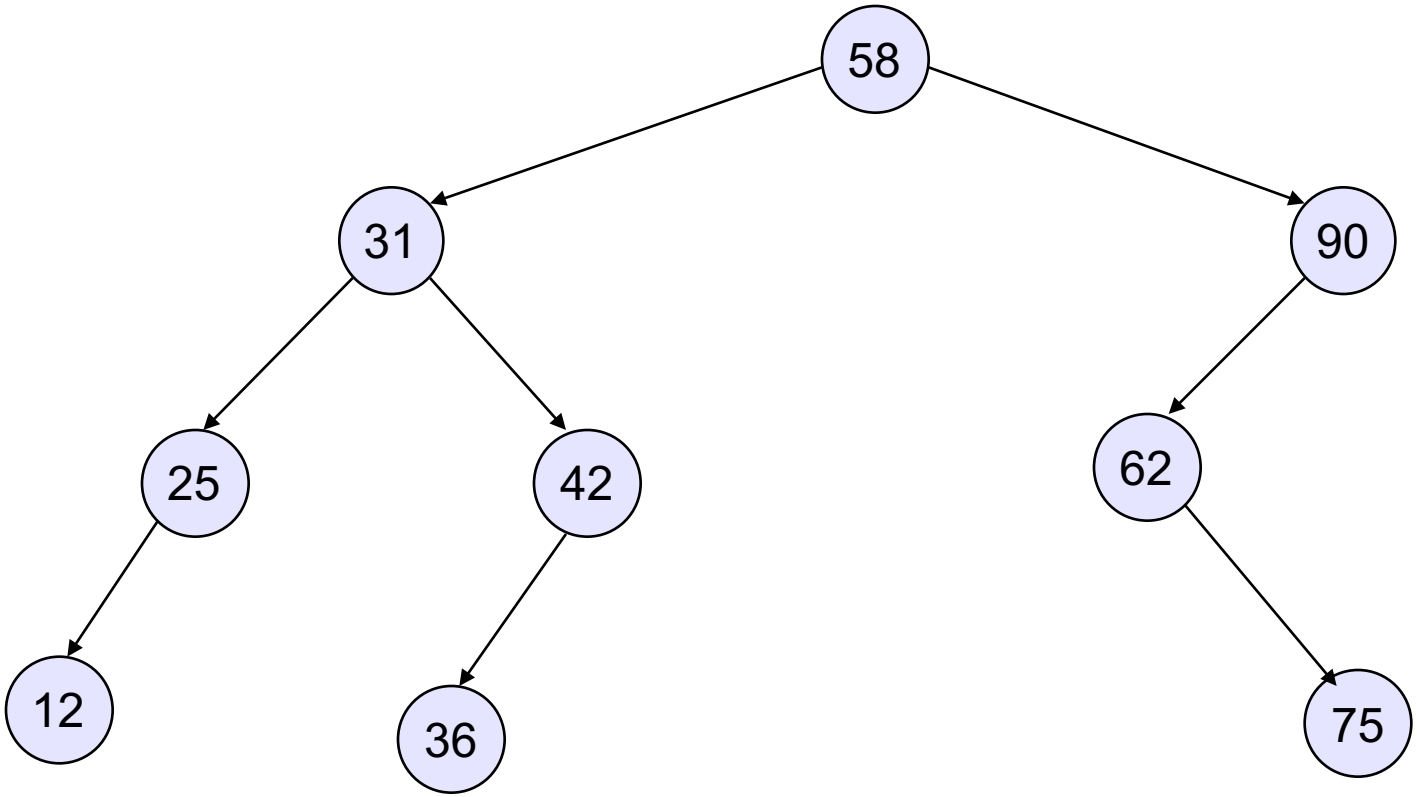
Searching for the element 36

Binary Search Trees



Searching for the element 36

Binary Search Trees



36=36 - success

Searching for the element 36

Computational Cost of Binary Tree Searching

- The binary tree search algorithm executes a constant number of operations for each node during the traversal.
- The binary search algorithm starts from the root and goes down one level at the time.
- The number of levels in a binary search tree is called the height *h*.
- The method findElement runs in O(*h*) time. This can potentially be a problem as *h* can potentially be close to *n*. Thus, it is essential to keep the tree height optimal (as close to $O(\log n)$ as possible). The way to achieve this is to balance a tree after each insertion (AVL trees).

Dictionary Search Using a Binary Search Tree

- The method findElement(*k*) can be performed on a dictionary *D* if we store *D* as a binary search tree and call the method TreeSearch(*k, T.*root()).

Algorithm TreeSearch(*k,v*);

Input: A search key *k* and a node *v* of a binary search tree;

Output: A node *w* of *T* equal to *k*, or an exception;

if *k*=key(*v*) **then return** *v*;

else if *k* is an external node **then return** NO_SUCH_KEY;

else if *k*<key(*v*) **then return** TreeSearch(*k, T.*leftChild(*v*));

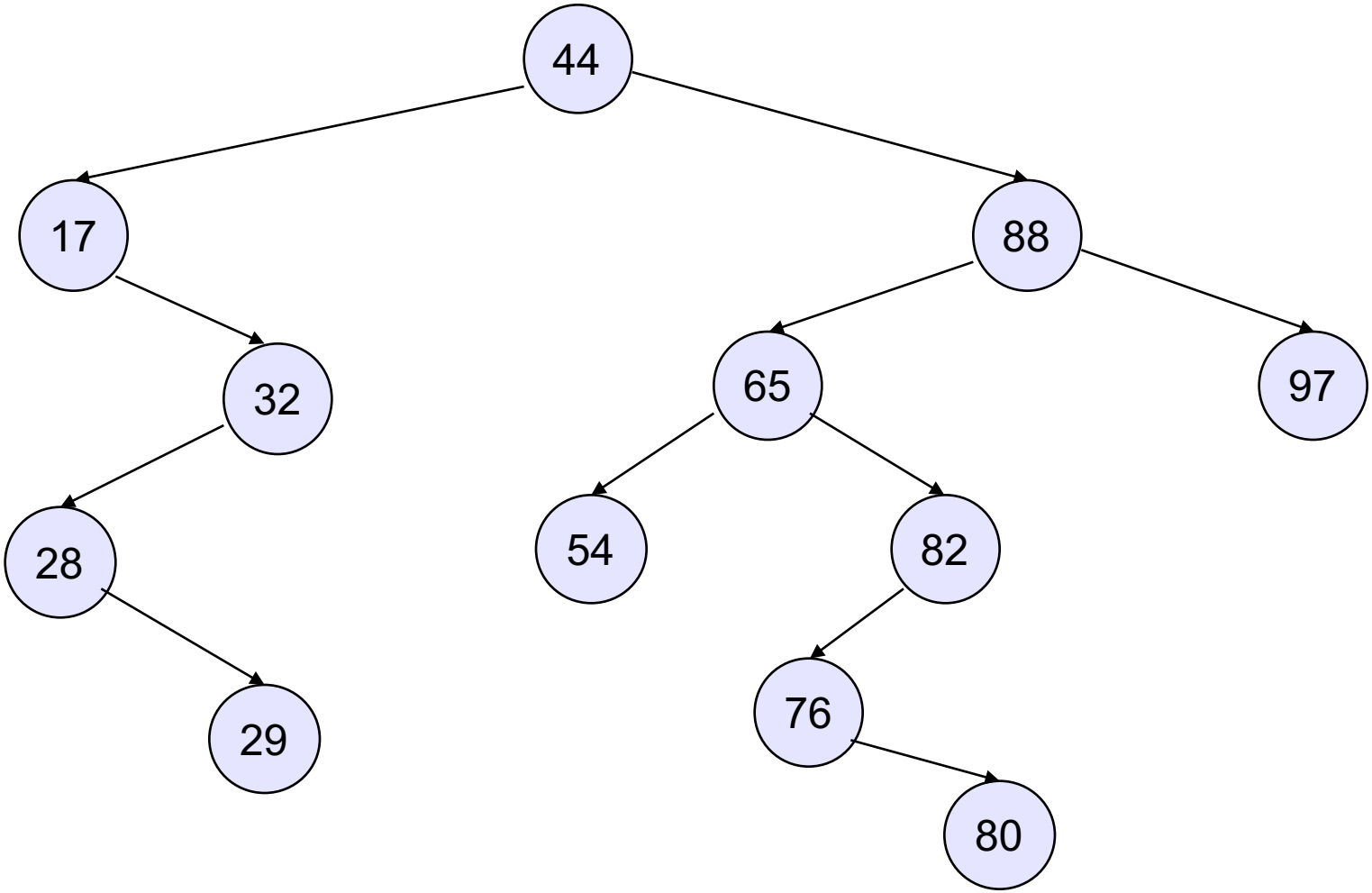
else return TreeSearch(*k, T.*rightChild(*v*));

end if

Insertion into a Binary Search Tree

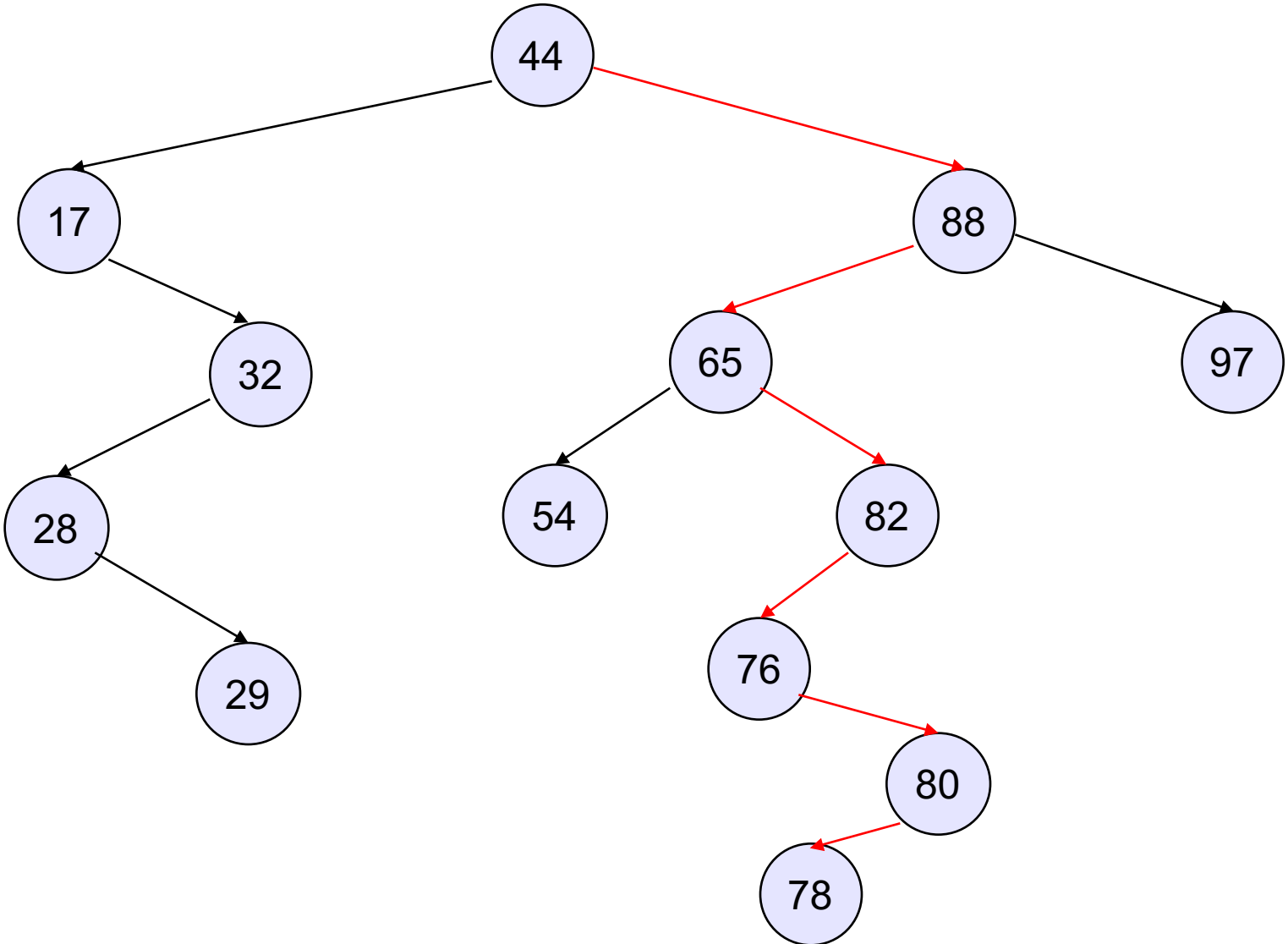
- To perform the operation insertElem(*k,e*) into a dictionary *D* implemented as a binary search tree, we call the method TreeSearch(*k,T.*root()). Suppose that *w* is the node returned by TreeSearch. Then:
  - If besides *w* a flag NO_SUCH_KEY is returned, then compare *e* with *w.* If *e*<*w*, create a new left child and insert the element *e* with the key *k*. Otherwise, create a right child and insert the element *e* with the key *k*.
  - If only the node *w* is returned (there is another item with key *k*), we call the algorithm TreeSearch(*k,T.*leftChild(*w*)) and TreeSearch(*k,T.*rightChild(*w*)) and recursively apply the algorithm returned by the node TreeSearch.

Insertion into a Binary Search Tree



Insertion of an item with the key 78 into a binary search tree

Insertion into a Binary Search Tree



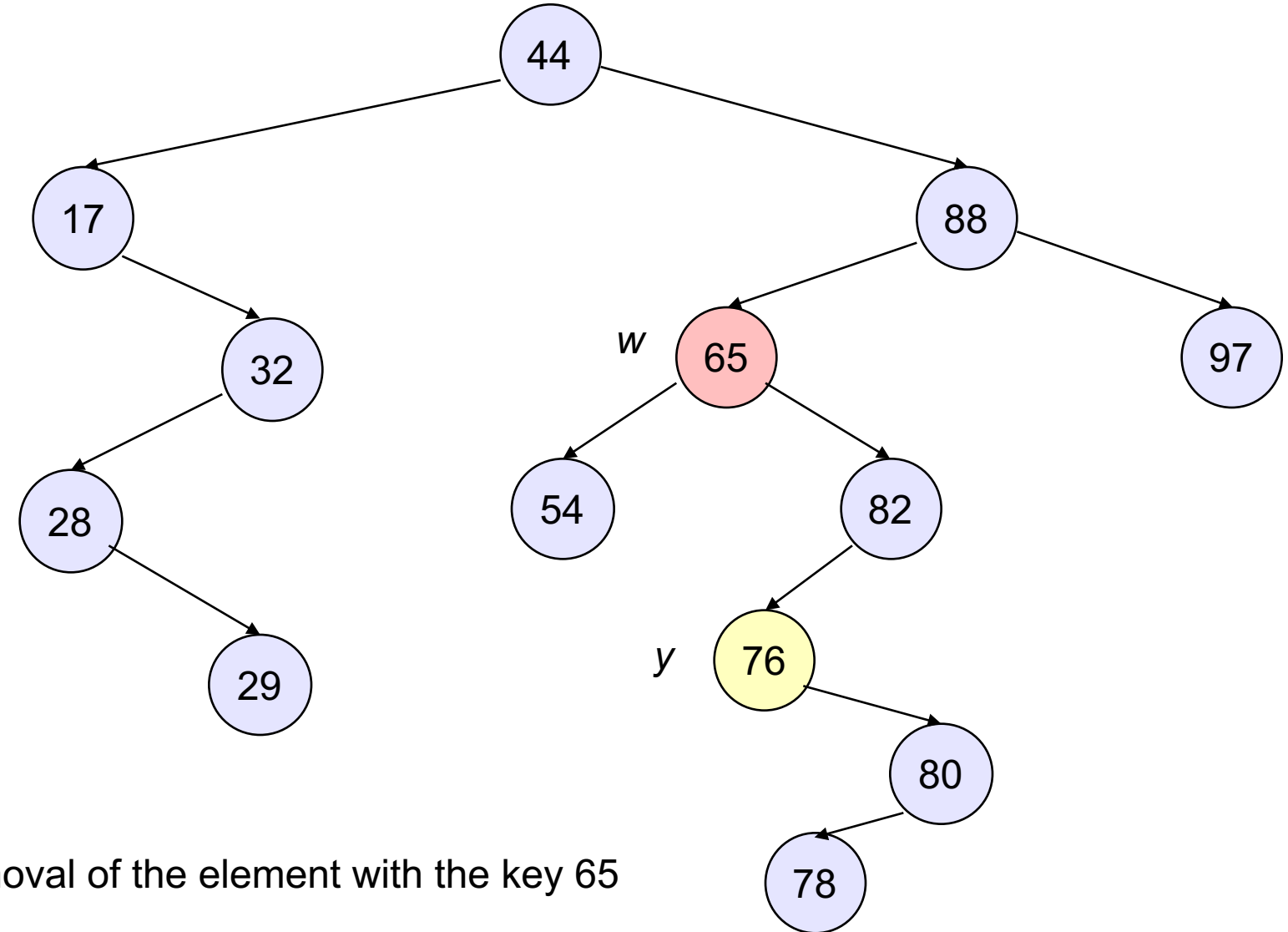
Removal from a Binary Search Tree

- Performing removeElement(*k*) on a dictionary *D* implemented with a binary search tree introduces an additional difficulty that the tree needs to remain connected after the removal.
- We need to execute first TreeSearch(*k,T.*root()) to find a node with a key *k*. If the algorithm returns an exception, there is no such element in *D.* If the key *k* is found in *D*, we distinguish two cases:
  - If the node with the key *k* is the leaf node, the removal operation is simple;
  - If the node with the key *k* is an internal node, its simple removal would create a hole. To avoid this, we need to do the following:

Removal from a Binary Search Tree

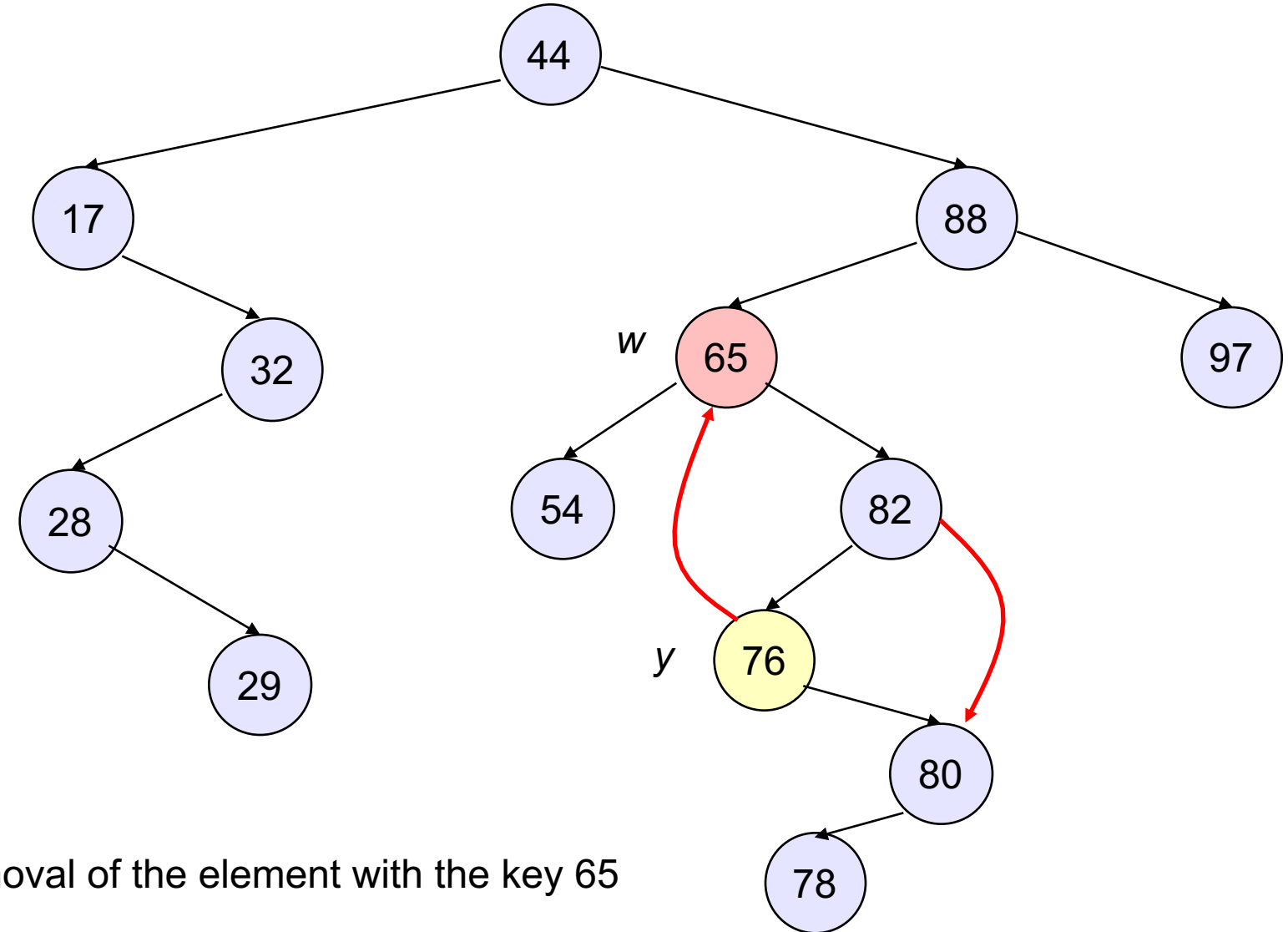
1. Find the first node *y* that follows *w* in an inorder traversal. It is the leftmost internal node in the right subtree of *w* (go right from *w*, and then follow the left children;
2. Save the element stored at *w* into a temporary variable *t*, and move *y* into *w*. This would remove the previously stored element of *w*;
3. Remove the element *y* from *T*;
4. Return the element stored in a temporary variable *t*;

Removal from a Binary Search Tree



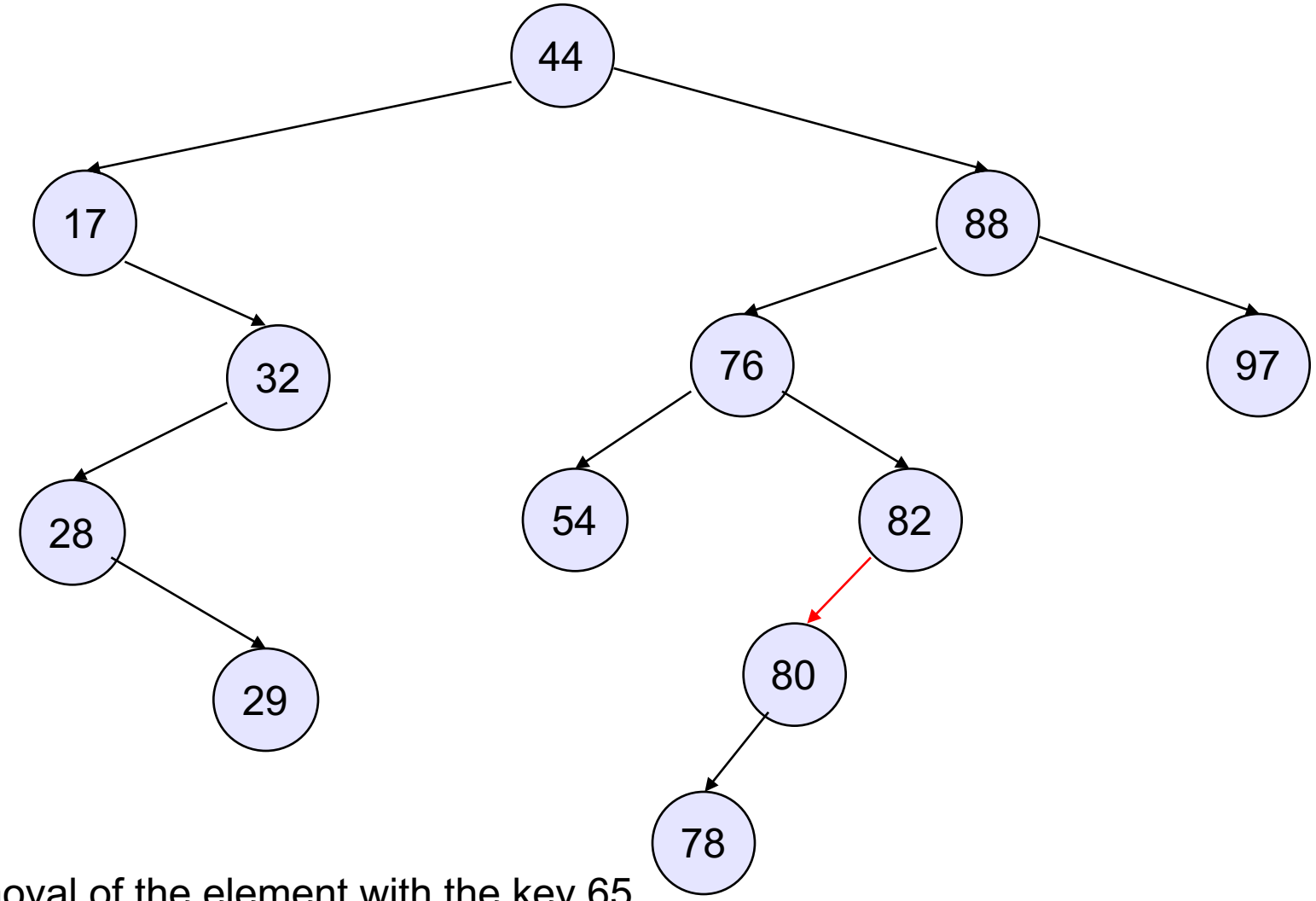
Removal of the element with the key 65

Removal from a Binary Search Tree



Removal of the element with the key 65

Removal from a Binary Search Tree



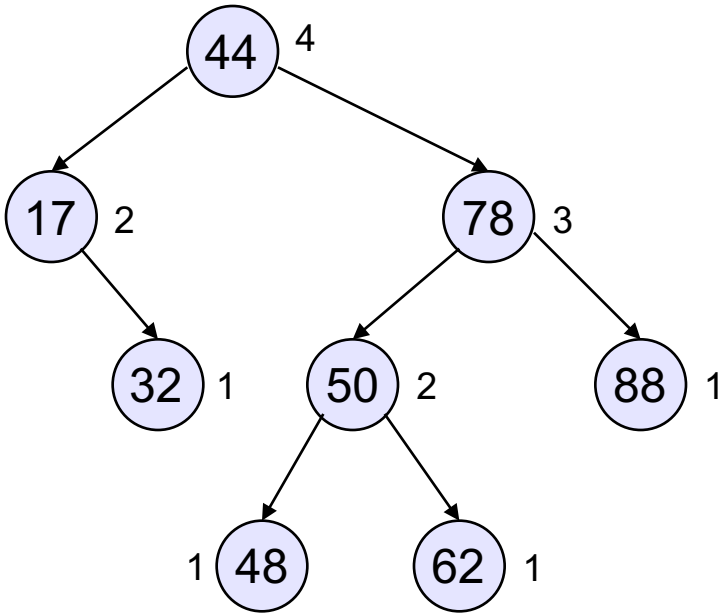
Removal of the element with the key 65

AVL Trees

- The idea behind introducing the AVL trees is to improve the efficiency of the basic operations for a dictionary.
- The main problem is that if the height of a tree that implements a dictionary is close to *n*, the basic operations execute in time that is asymptotically no better than that obtained from the dictionary implementations via log files or lookup tables.
- A simple correction is to have an additional property added to the definition of a binary search tree to keep the logarithmic height of the tree. This is the height balance property:

  For every internal node *v* of the tree *T*, the heights of its children can differ by at most 1.

AVL Trees

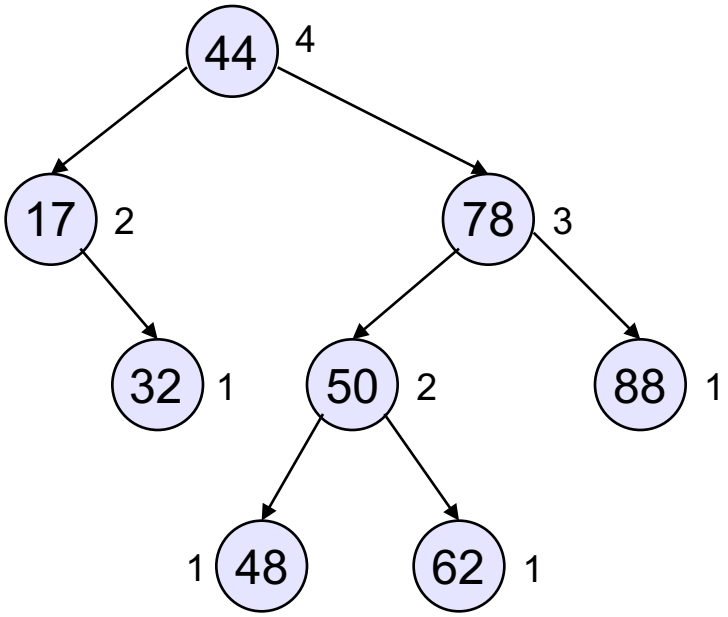


An example of an AVL tree with the node heights

- Any subtree of an AVL tree is an AVL tree itself.
- The height of an AVL tree that stores *n* items is $O(\log n)$.
- This implies that searching for an element in a dictionary implemented as an AVL tree runs in $O(\log n)$ time.

Insertion into an AVL Tree

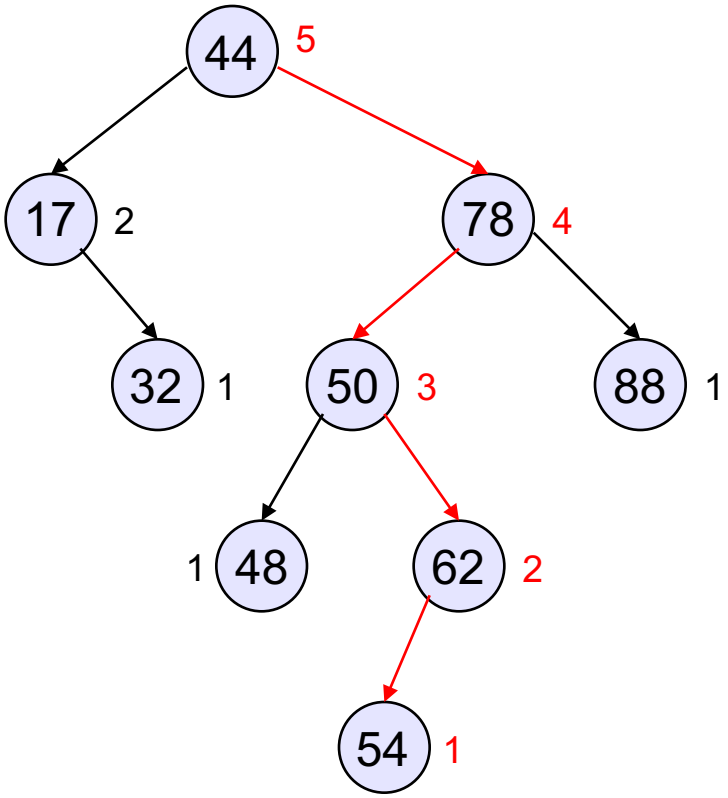
- The first phase of an element insertion into an AVL tree is the same as for any binary tree.



An example of inserting the element with the key 54 into an AVL tree

Insertion into an AVL Tree

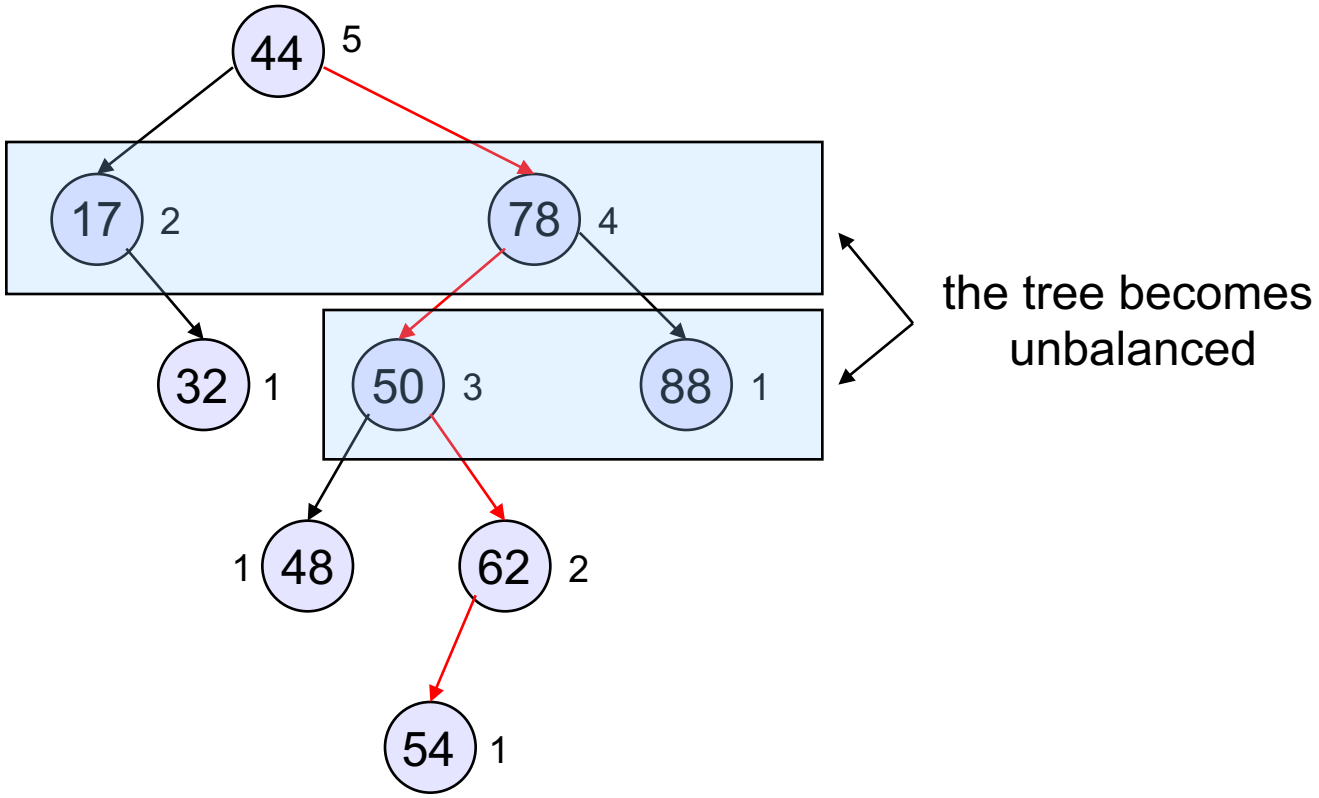
- The first phase of an element insertion into an AVL tree is the same as for any binary tree.



An example of inserting the element with the key 54 into an AVL tree

Insertion into an AVL Tree

- The first phase of an element insertion into an AVL tree is the same as for any binary tree.

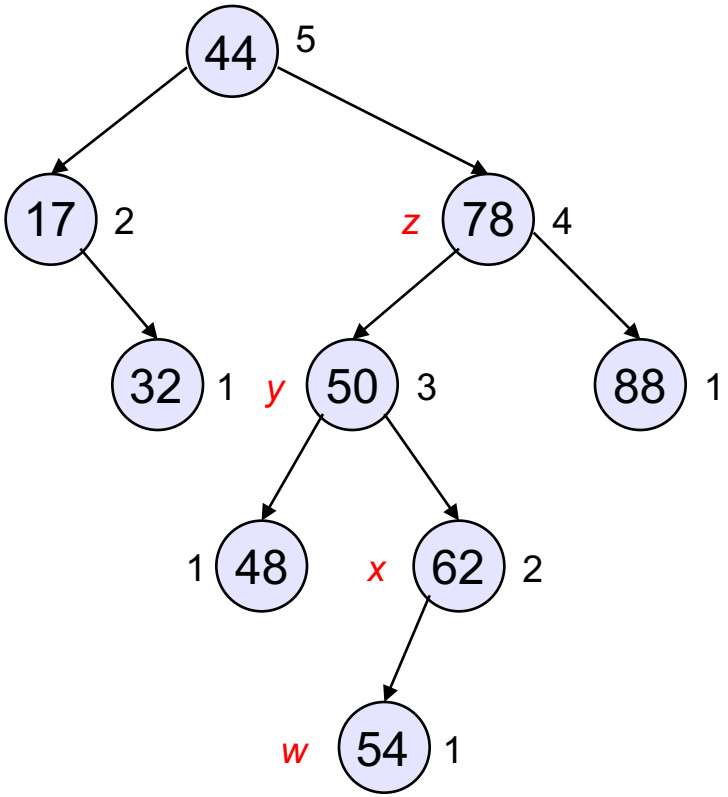


An example of inserting the element with the key 54 into an AVL tree

Insertion into an AVL Tree

- Suppose that the tree satisfies the height-balance property prior to the insertion of a new element *w*. After inserting the node *w*, the heights of all nodes that are on the path from the root to the newly inserted node will increase. Consequently, these are the only nodes that may become unbalanced by the insertion.
- We restore the balance of the nodes in the AVL tree by a search and repair strategy.
- Let *z* be the first node on the path from *w* to root that is unbalanced.
- Denote by *y* the child of *z* with a larger height (if there is a tie, choose *y* to be an ancestor of *z*).
- Denote by *x* the child of *y* with a larger height (if there is a tie, choose *x* to be an ancestor of *z*).

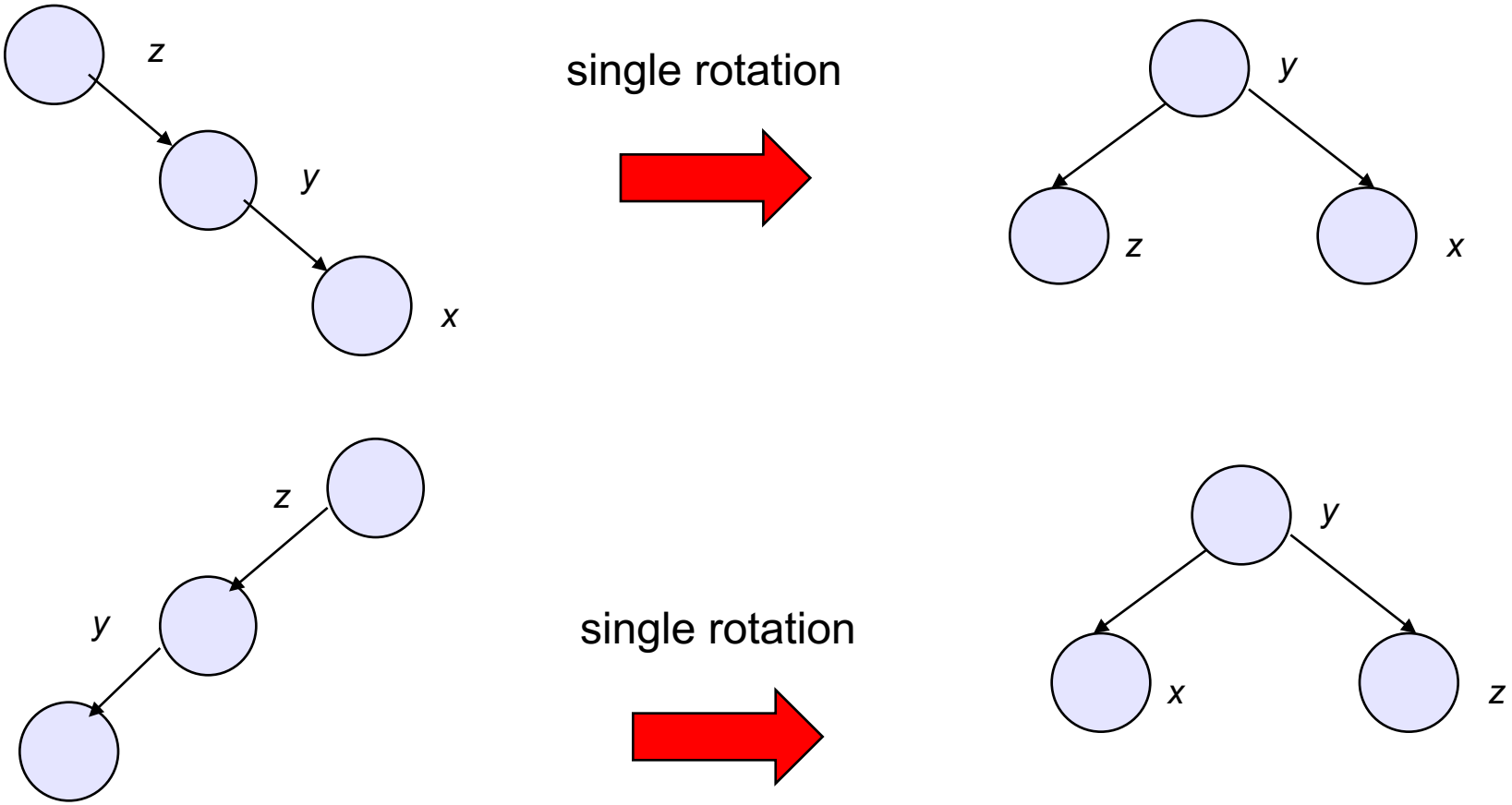
Insertion into an AVL Tree



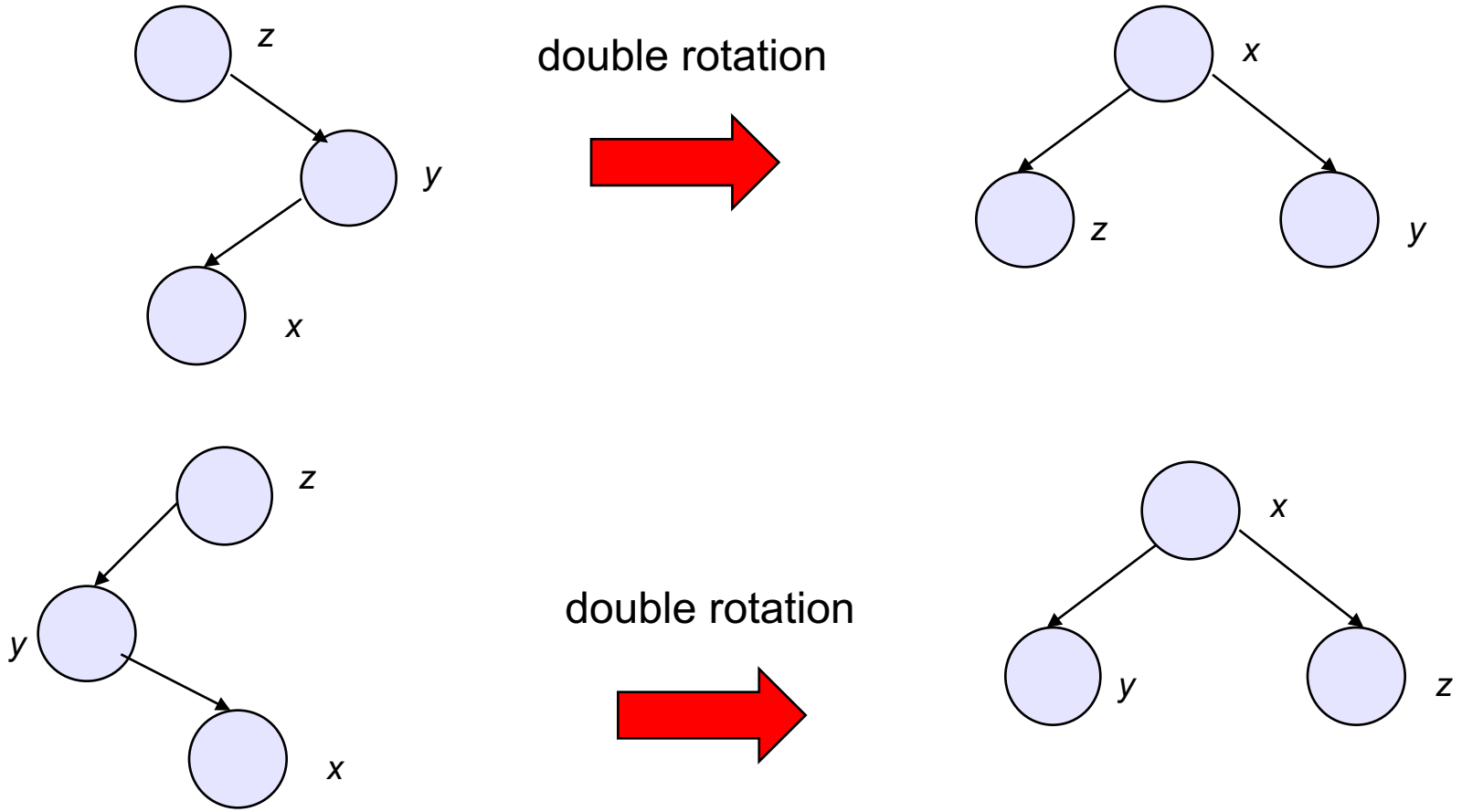
Balancing an AVL Tree

- The node *z* becomes unbalanced because of an insertion into the subtree rooted at its child *y*.
- The subtree rooted at *z* is rebalanced by the trinode restructuring method. There are 4 cases of the restructuring algorithm. The modification of a tree *T* by a trinode restructuring operation is called a rotation. The rotation can be single or double.
- The trinode restructuring methods modify parent-child relationships in O(1) time, while preserving the inorder traversal ordering of all nodes in *T*.

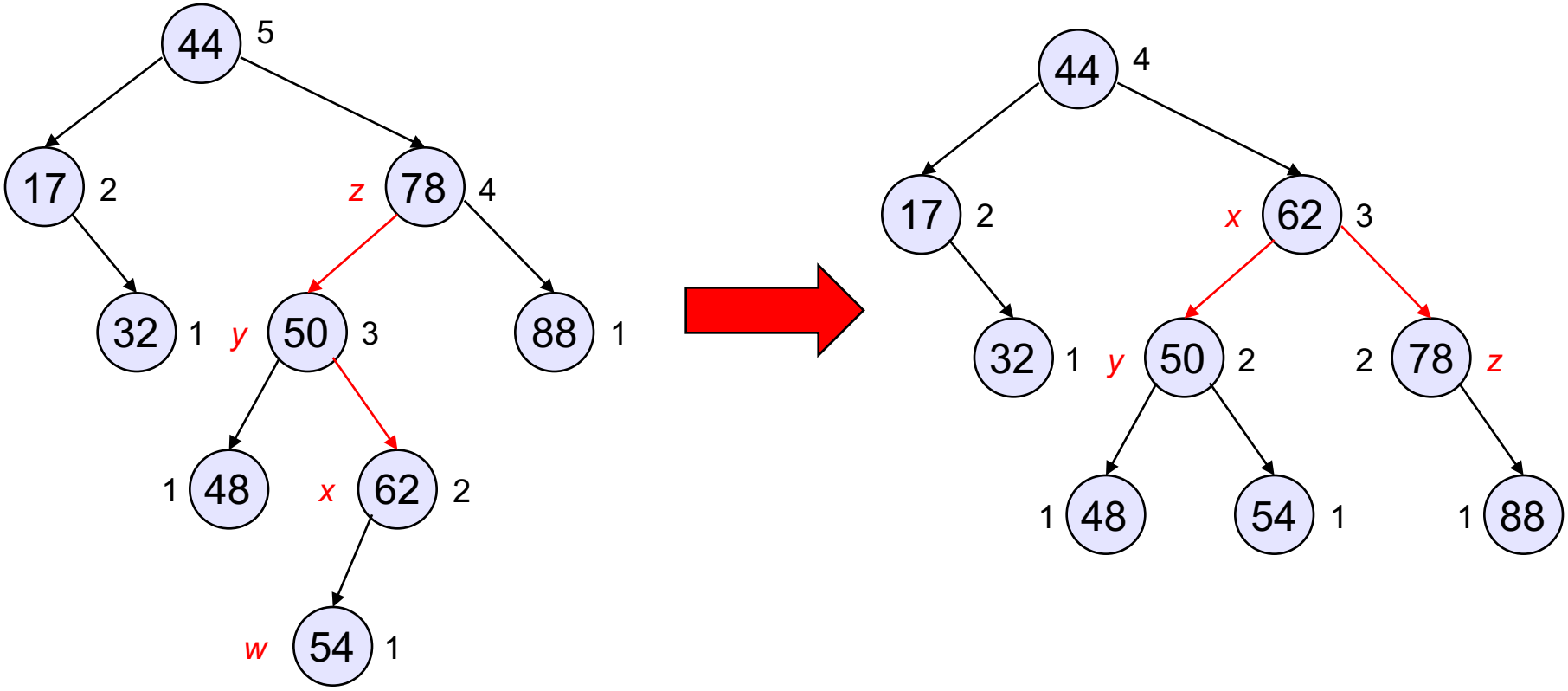
Trinode Restructuring by Single Rotation



Trinode Restructuring by Double Rotation



Balancing an AVL Tree



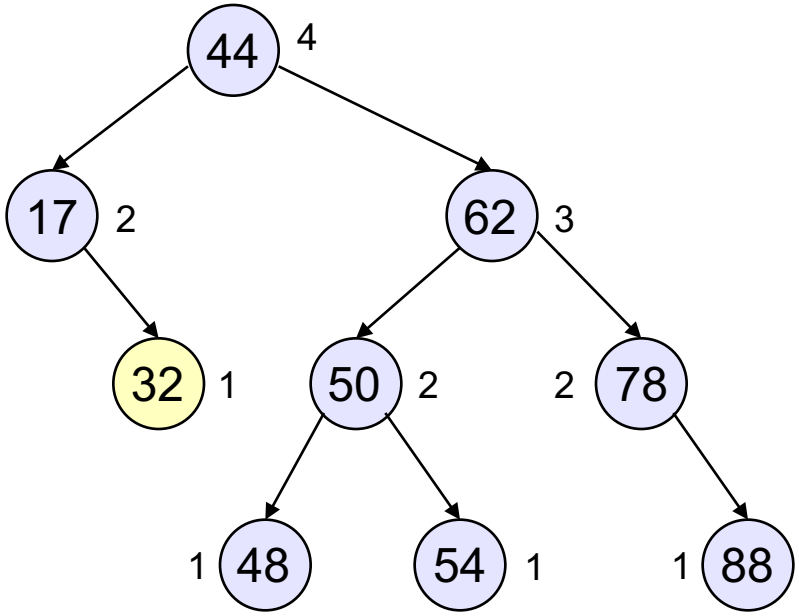
Removal from an AVL Tree

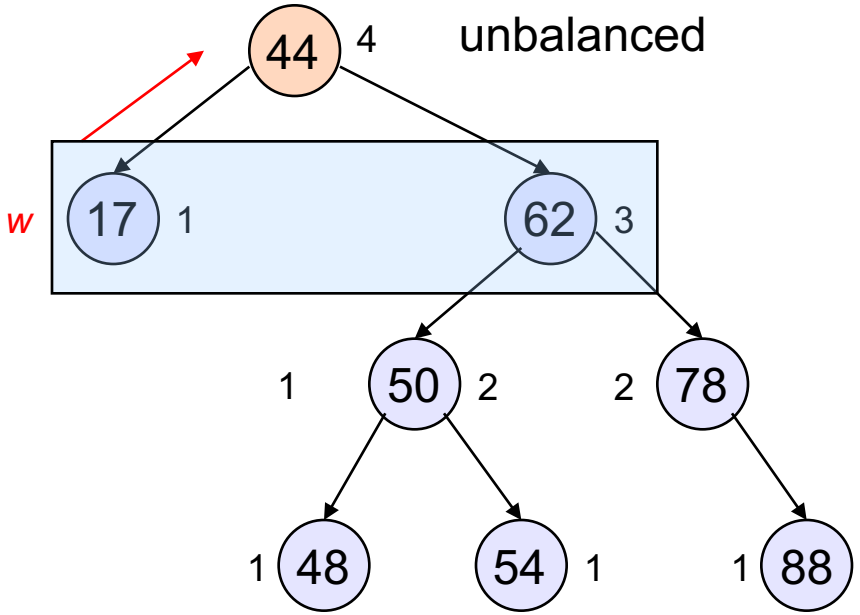
- The first phase of the element removal from an AVL tree is the same as for a regular binary search tree. This can, however, violate the height-balance property of an AVL tree.
- If we remove an external node, the height balance property will be satisfied.
- But, if we remove an internal node and elevating one of its children into its place, an unbalanced node in T may occur. This node will be on the path from the parent w of the previously removed node to the root of T.
- We use the trinode restructuring after the removal to restore the balance.

Removal from an AVL Tree

- Let *z* be the first node encountered going upwards from *w* (the parent of the removed node) towards the root of *T*.
- Let *y* be the child of *z* with a larger height (i.e. it is a child of *z*, but not an ancestor of *w*).
- Let *x* be a child of *y* with a larger height (this choice may not be unique).
- The restructuring operation is then performed locally, by restructuring a subtree rooted in *z.* This may not recover the height balance property, so we need to continue marching up the tree and looking for the nodes with no height balance property.
- The operation complexity of the restructuring is proportional to the height of a tree, hence $O(\log n)$.

Removal from an AVL Tree





Removal of the element with key 32 from the AVL tree

Removal from an AVL Tree