COMP 8547 Advanced Computing Concepts

Course instructor: Dr. Olena Syrotkina

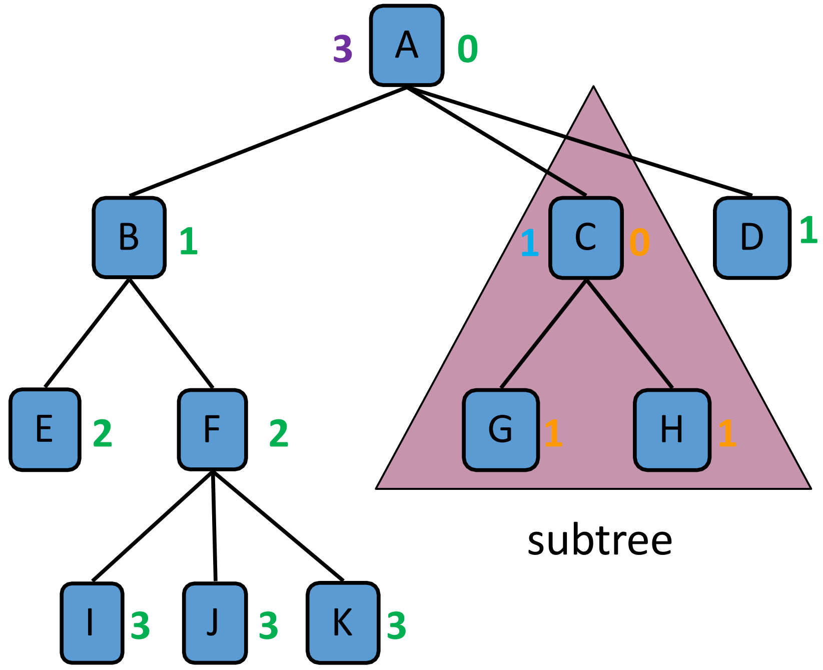
Chapter 3 - Search Trees

Contents

Trees - definition, terminology

- Graph theory: A tree T is an undirected acyclic graph
- A tree can be either unrooted or rooted. This chapter covers rooted trees.
- Informally: A tree T is a set of nodes, which have a parent-child relationship.
- Subtree: tree consisting of a node of T and its descendants
- Properties:
  - If T is nonempty, it has a single node called the root of T
  - Each node v of T (except the root) has a unique parent
  - Every node v with parent w is a child of w
  - A tree T can be empty

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (or leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree (or a subtree): maximum depth of any node
- Descendant of a node: child, grandchild, grand-grandchild, etc.

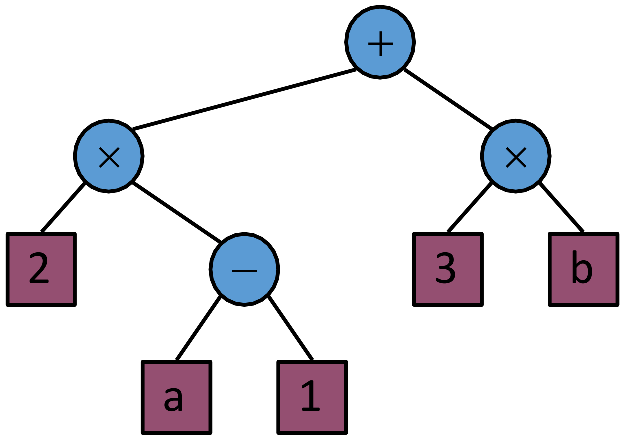


Binary trees

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for **proper** binary trees)
  - The children of a node are in an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - an empty tree,
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

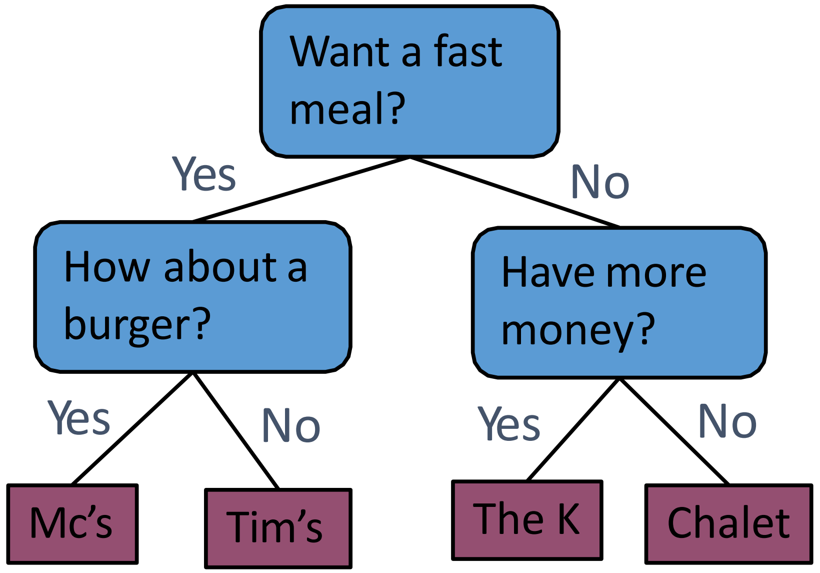
Arithmetic expressions

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $((2 \times (a - 1)) + (3 \times b))$



Decision trees

- Each internal node is associated with an attribute or variable
- Each external node corresponds to an outcome
- Example: how to decide on a restaurant



Proper binary trees - properties

- Notation

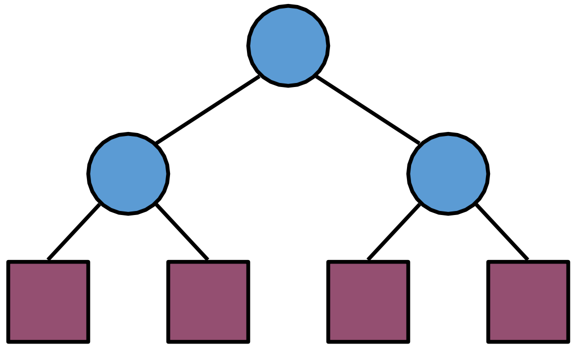
  $\boldsymbol{n}$ number of nodes

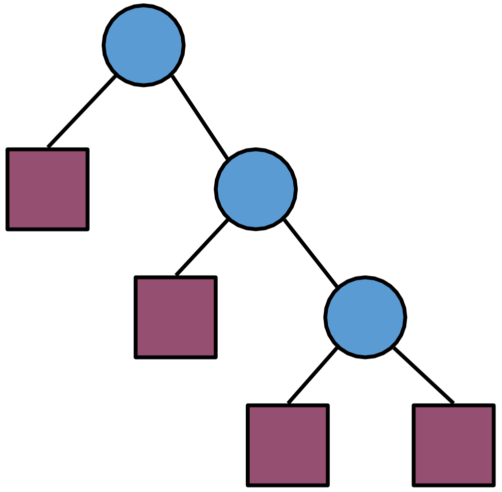
  $\boldsymbol{e}$ number of external nodes

  $\boldsymbol{i}$ number of internal nodes

  $\boldsymbol{h}$ height

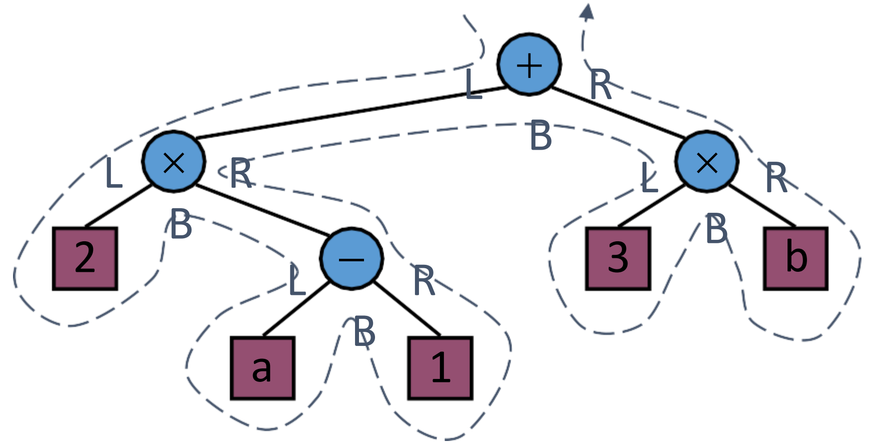
- Properties:
  - $\boldsymbol{n} = \boldsymbol{e} + \boldsymbol{i}$
  - $\boldsymbol{e} = \boldsymbol{i} + 1$
  - $\boldsymbol{n} = 2\boldsymbol{e} - 1$
  - $\boldsymbol{h} + 1 \leq \boldsymbol{e} \leq 2^{\boldsymbol{h}}$
  - $\boldsymbol{h} \leq \boldsymbol{i} \leq 2^{\boldsymbol{h}} - \mathbf{1}$
  - $\log(\boldsymbol{n} + 1) - 1 \leq \boldsymbol{h} \leq (\boldsymbol{n} - 1)/2$





Binary trees – traversal

- **Preorder** (L): A node is visited before its left and right subtrees
- **Inorder** (B): A node is visited after its left subtree and before its right subtree
- **Postorder** (R): A node is visited after its left and right subtrees
- Euler tour traversal:
  - Recursively called on the left and right children
  - Visit every internal node three times
  - External node visited only once



```
Algorithm EulerTour(v)
  visit(v)
  if hasLeft(v)
      EulerTour(left(v))
  visit(v)
  if hasRight(v)
      EulerTour(right(v))
  visit(v)
```

- Ex. inorder: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v
- Ex. Inorder: Print arithmetic expression
  - It outputs:

    $((2 \times (a - 1)) + (3 \times b))$

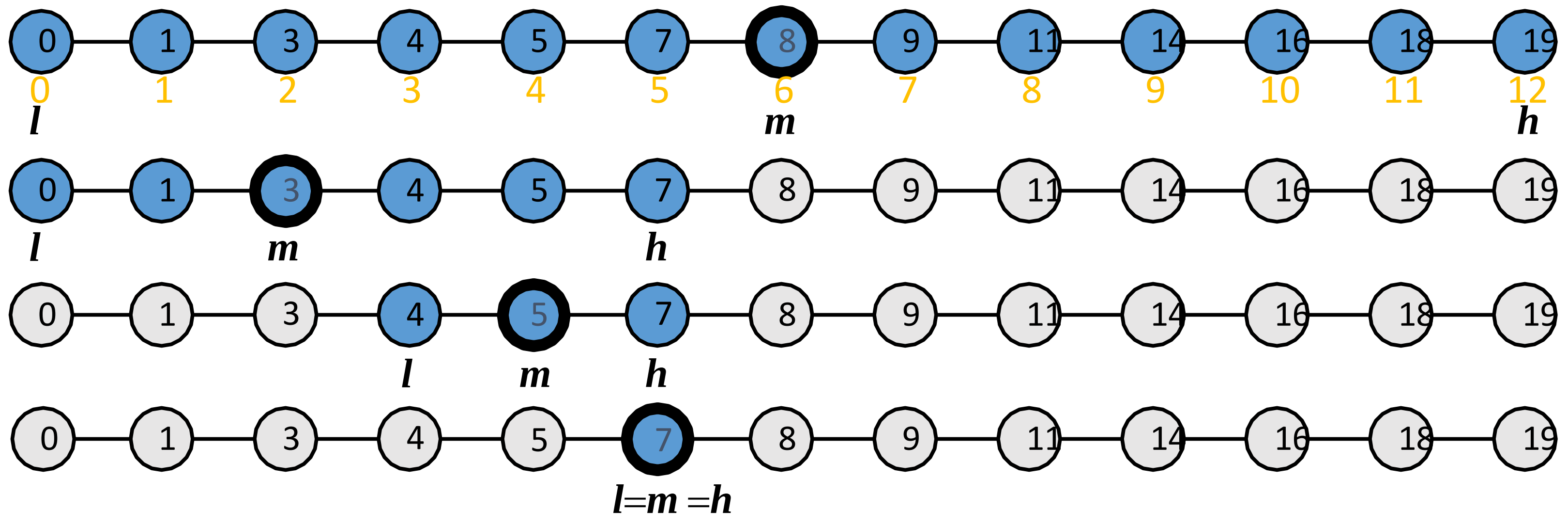
- Ex. Postorder: evaluate arithmetic expression
  - Specialization of postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

```
Algorithm evalExpr(v)
  if isExternal (v)
     return v.element ()
  else
     x ← evalExpr(left (v))
     y ← evalExpr(right (v))
     ◊ ← operator stored at v
  return x ◊ y
```

Sorted maps - binary search

- Binary search performs find(k) in a sorted map implemented on an array, sorted by key
  - Performs a search in a logarithmic number of steps: O(log n)
- Can be easily adapted to find more than one entry with key k
- Can also find keys in a range in O(log n)
- Example: find(7)

```
Algorithm binarySearch(S, k, low, high):
if low > high then
        return null
else
    mid ← ⌊(low + high) / 2⌋
    e ← S.get(mid)
    if k = e.getKey() then
        return e
    else if k < e.getKey() then
            return binarySearch(S, k, low, mid-1)
        else
            return binarySearch(S, k, mid+1, high)
```



Binary search trees (BST)

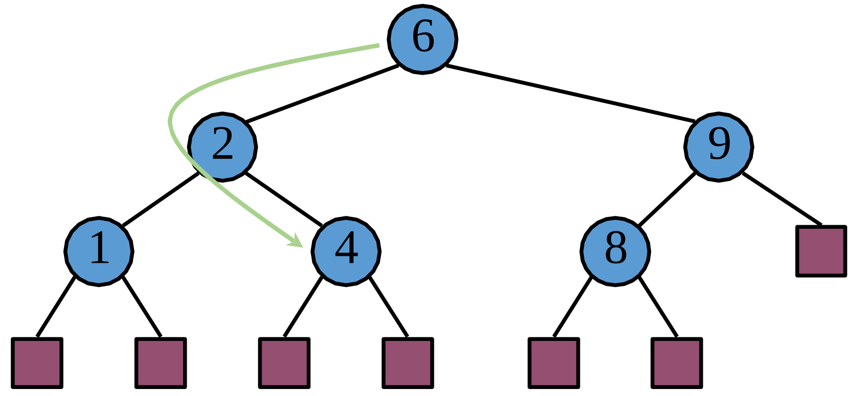
- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \le key(v) \le key(w)$
- External nodes do not store items
- BST are proper binary trees
- Inorder traversal of a binary search tree visits the keys in increasing order
- A binary search tree implements a sorted map ADT

Search

- Can be easily implemented recursively

```
Algorithm TreeSearch(k, v)
  if T.isExternal (v)      {base case}
    return v
  if k < key(v)            {general case}
    return TreeSearch(k, T.left(v))
  else if k > key(v)
    return TreeSearch(k, T.right(v))
  else              {k = key(v)   -- base case}
    return v
```

- Example: Find 4

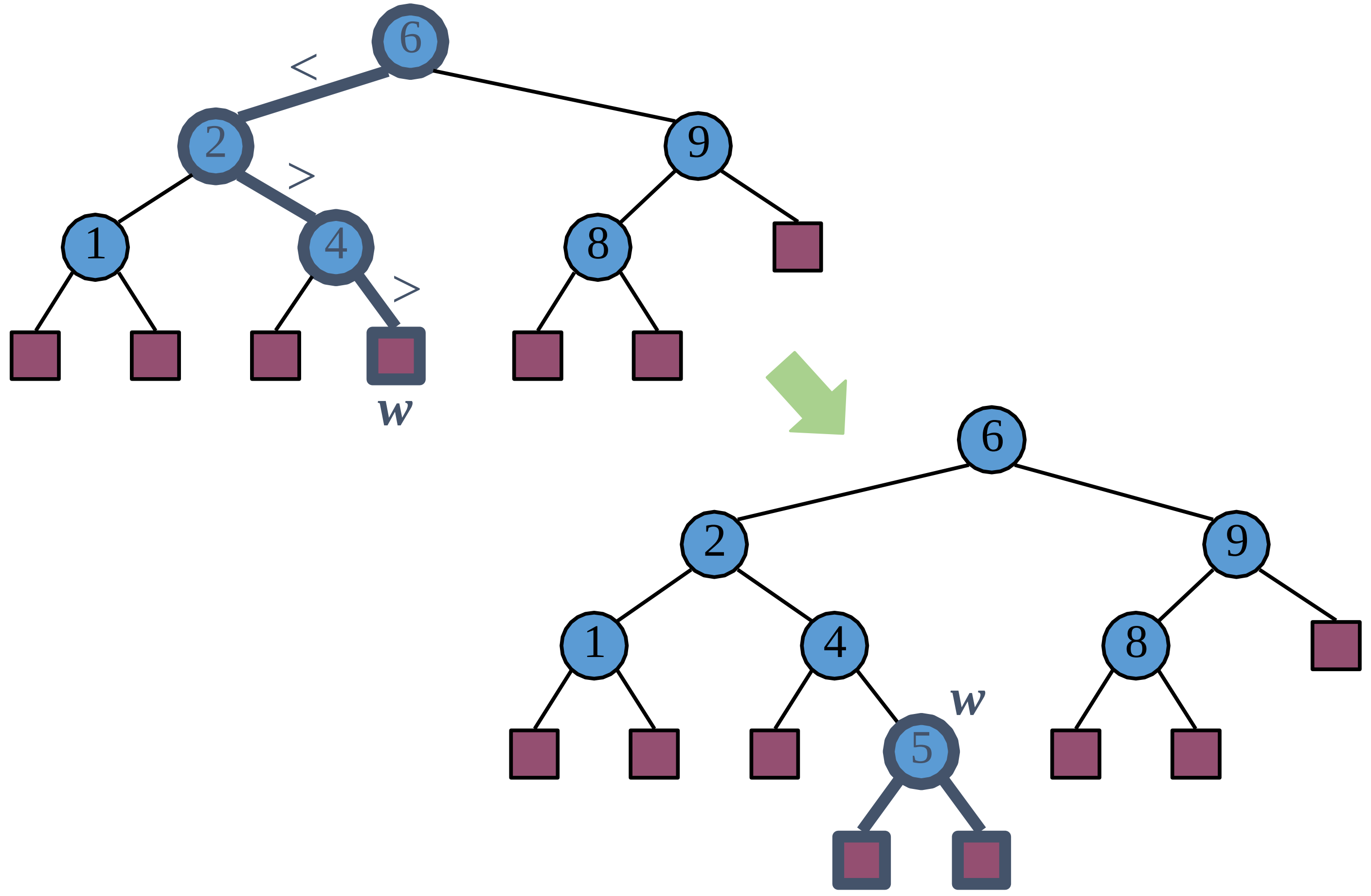


Binary search trees - insertion

- Search for key k using TreeSearch
  - If k is not found, it is inserted at w, be the leaf reached by the search
  - If k is already in the tree in node w, the same insertion method is used with the left child of w

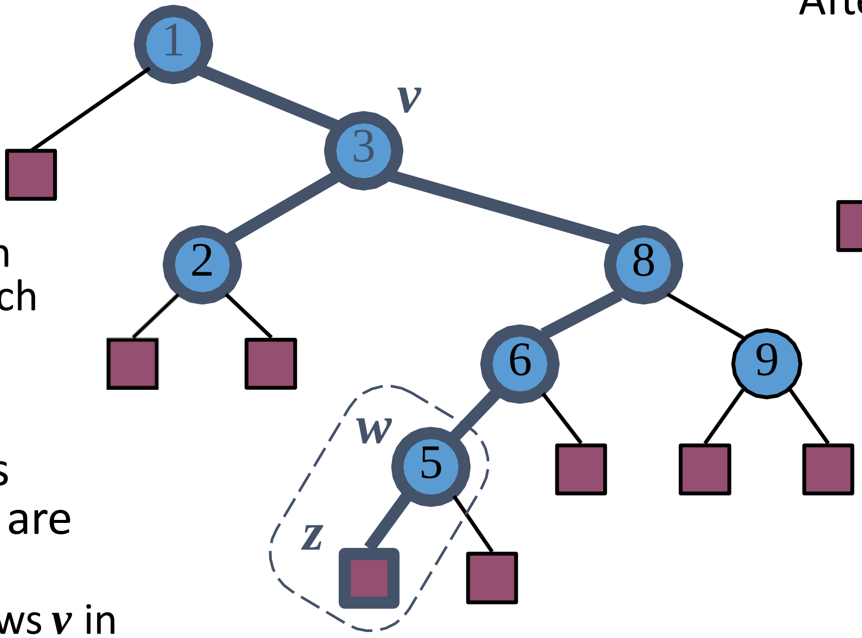
```
Algorithm TreeInsert(k,x,v)
Input: A key k, a value x, and
        a node v of T
Output: A new node w of T
w ← TreeSearch(k,v)
if k = key(w) then
    return
        TreeInsert(k,x,T.left(w))
T.insertAtExternal(k,x,w)
return w
```

- Example: insert 5
- Initial call:
  - TreeInsert(k,x,T.root())

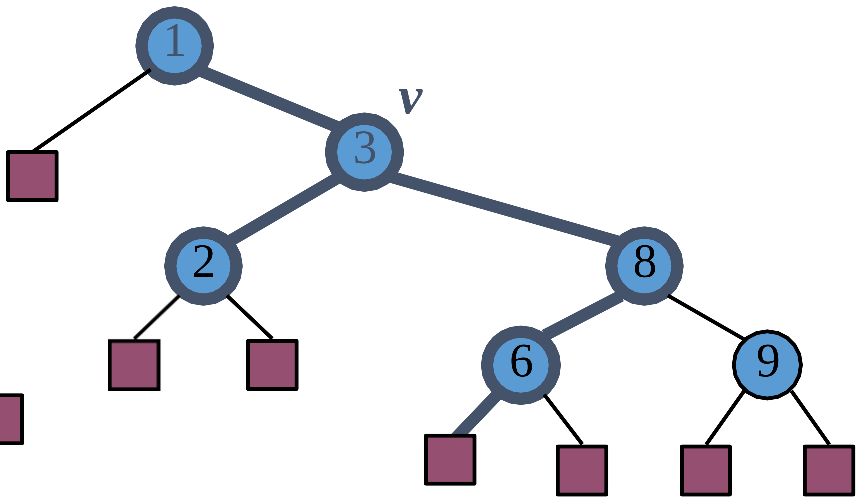


Binary search trees - deletion

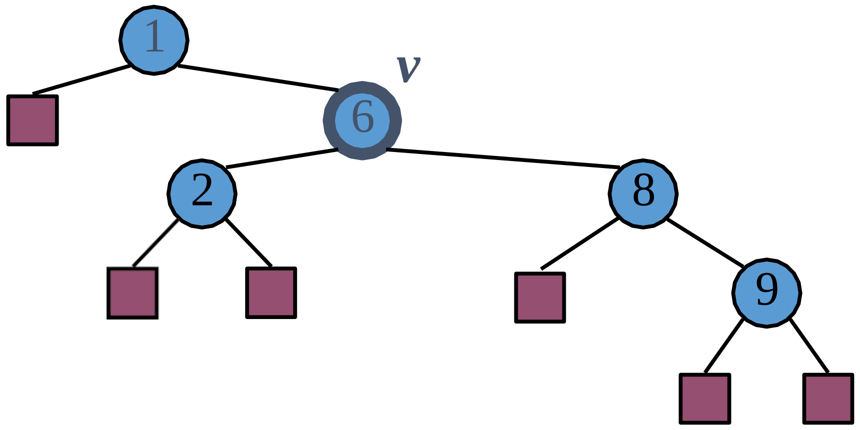
- First, we search for key ***k***
- Let ***v*** be the node storing ***k***
- **Case 1:** Node ***v*** has a leaf child ***w***
  - remove ***v*** and ***w*** from the tree with operation removeExternal(***w***), which removes ***w*** and its parent
  - Example: remove 5
- **Case 2:** The key ***k*** to be removed is stored at a node ***v*** whose children are both internal:
  - Find the internal node ***w*** that follows ***v*** in inorder traversal
  - Copy ***key***(***w***) into node ***v***
  - Remove node ***w*** and its left child ***z*** (which must be a leaf) by means of operation removeExternal(***z***)
  - Example: Remove 3



After removing 5:



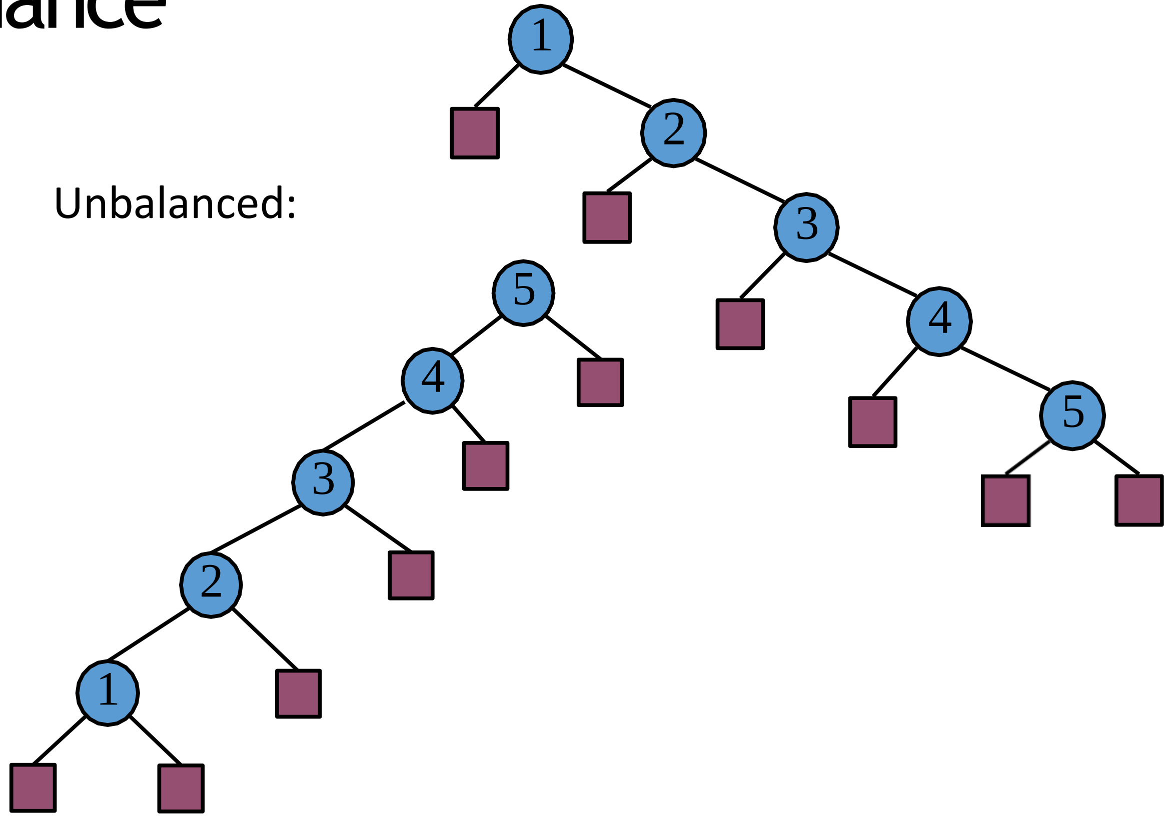
After removing 3:



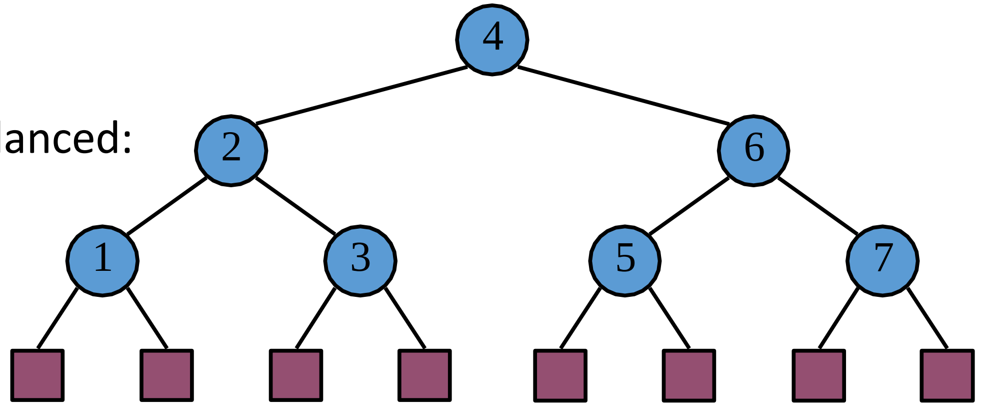
Binary search trees - performance

- Consider a sorted map with n entries implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - In the worst case, methods find, insert and remove run in $O(n)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case
- Average case:
  - Assume a random sequence of keys, and
  - all keys are equally likely
  - Then, the average depth of all nodes in the tree is O(log n)
  - However, average case of insertions/deletions may not be O(log n). At least no proof exists.

Unbalanced:



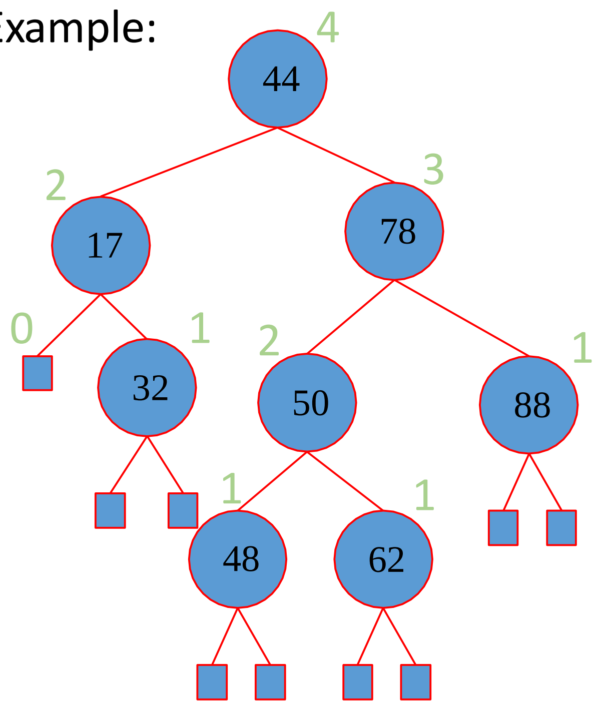
Balanced:



AVL trees

- **AVL trees are balanced**
- Def: An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the heights of the subtrees rooted at the children of v can differ by at most 1
- Named after its inventors:
  - Adel'son-Vel'ski and Landis
- **Fact**: The minimum number of keys of an AVL tree, n(h), is:
  $n(h) > 2^{h/2-1}$
- **Theorem**: The ***height*** of an AVL tree storing n keys is O(log n)
- Search in an AVL tree works as it does in a BST

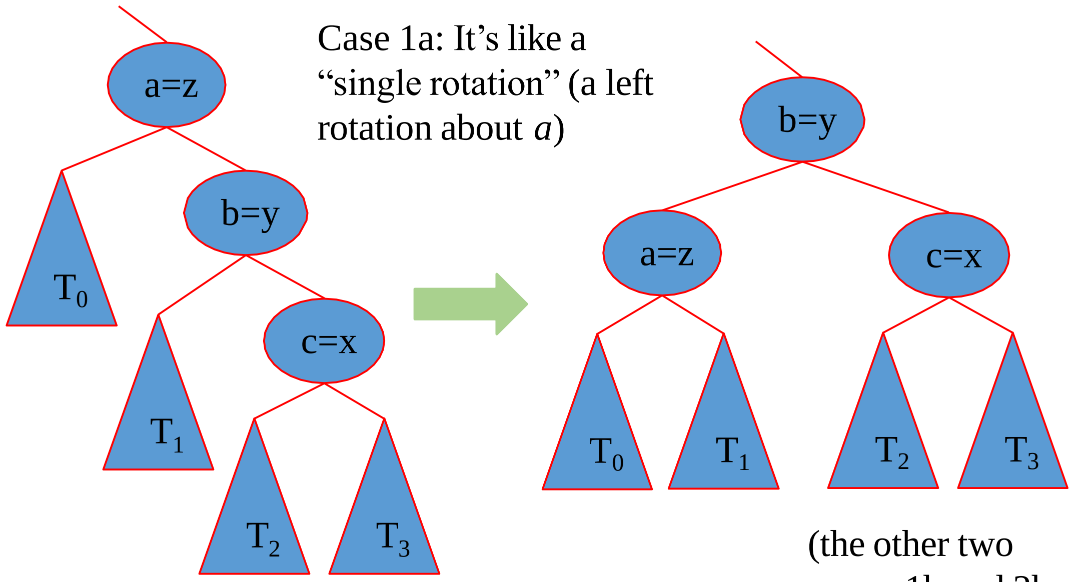
Example:



AVL trees - insertion

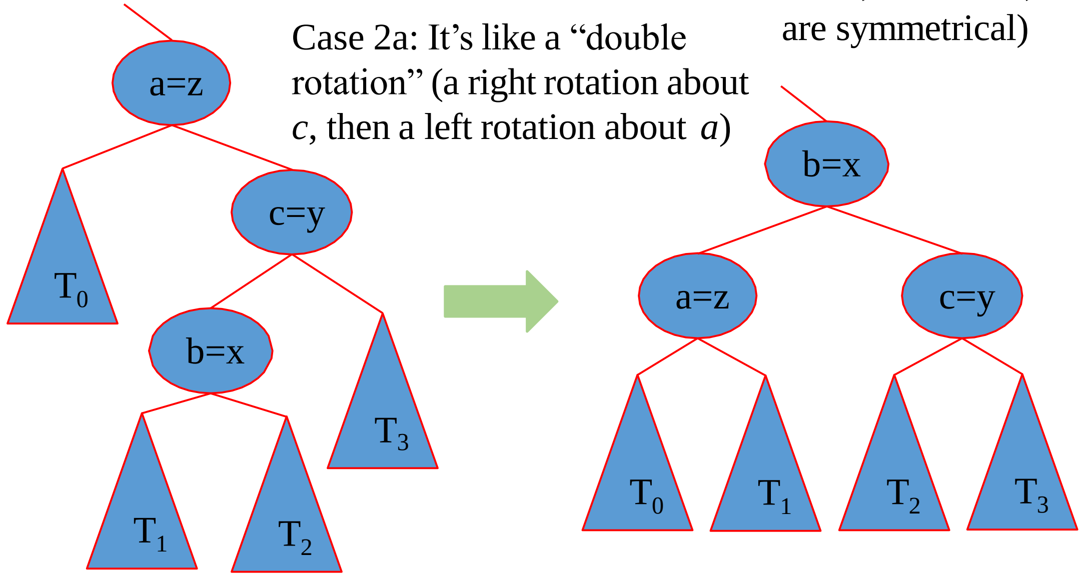
- Insertion is the same as in a BST
- But balance has to be restored
- After an insertion:
  - Trace a path from node inserted w all the way up to the root
  - For every node v, check that the heights of left and right children differ by at most 1
  - If for a node z, they differ by more than 1, then perform restructure(x):
    - ➢ Let y and x be the child and grandchild of z in that path
    - ➢ Apply the restructuring operation on z
  - Once one restructuring operation is applied **locally**, the AVL tree becomes balanced, **globally**
  - The heights of all nodes are stored and must be updated too.

Case 1a: It's like a "single rotation" (a left rotation about *a*)



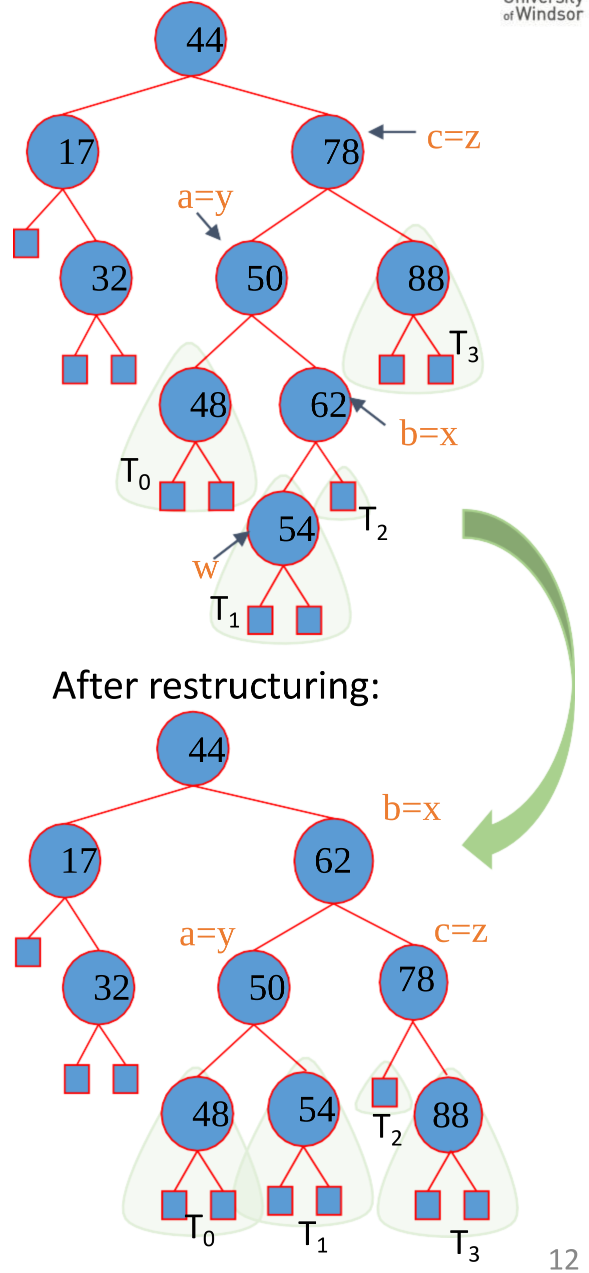
(the other two cases, 1b and 2b, are symmetrical)

Case 2a: It's like a "double rotation" (a right rotation about *c*, then a left rotation about *a*)



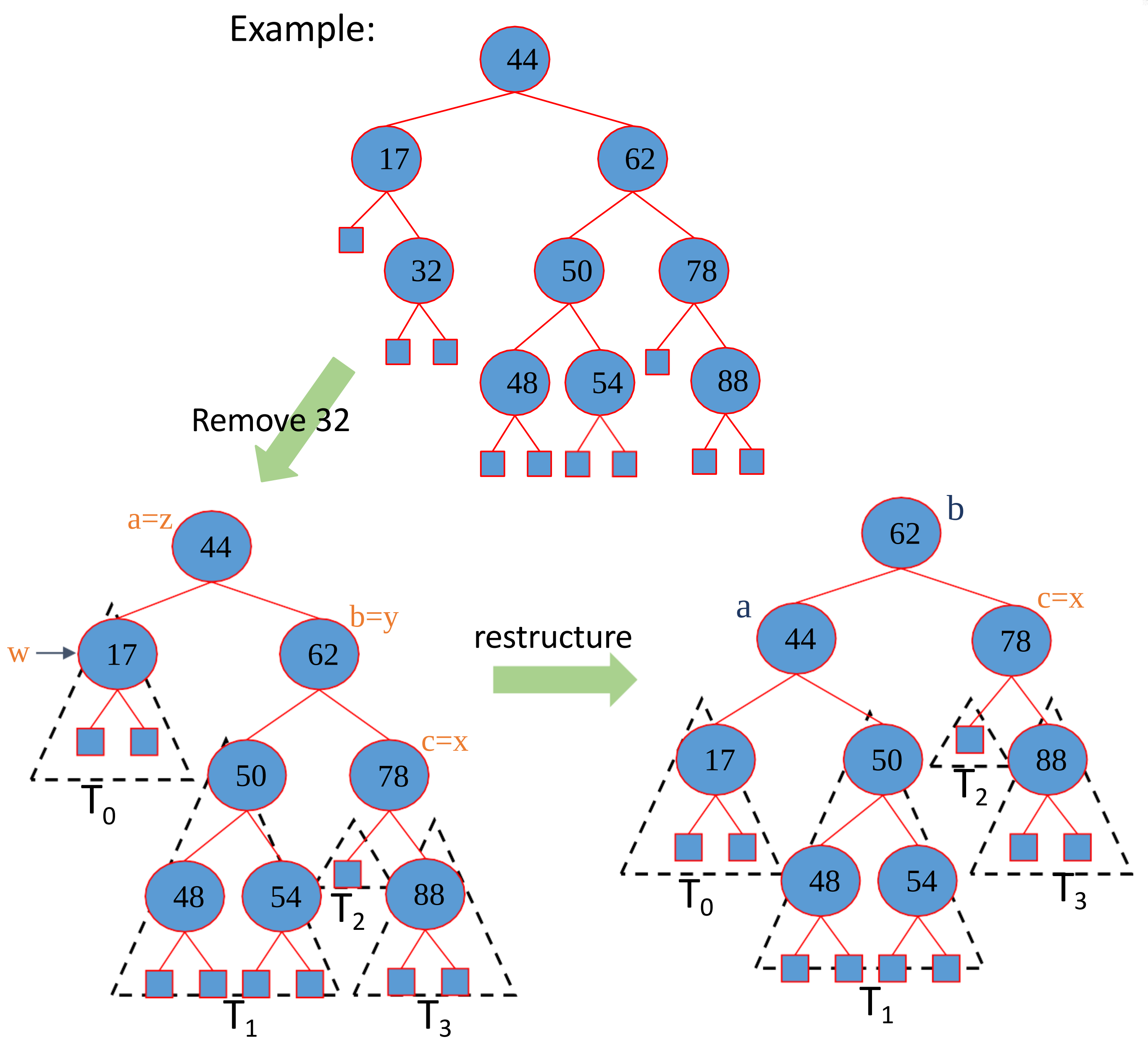
After insertion of 54:

After restructuring:



AVL trees - deletions

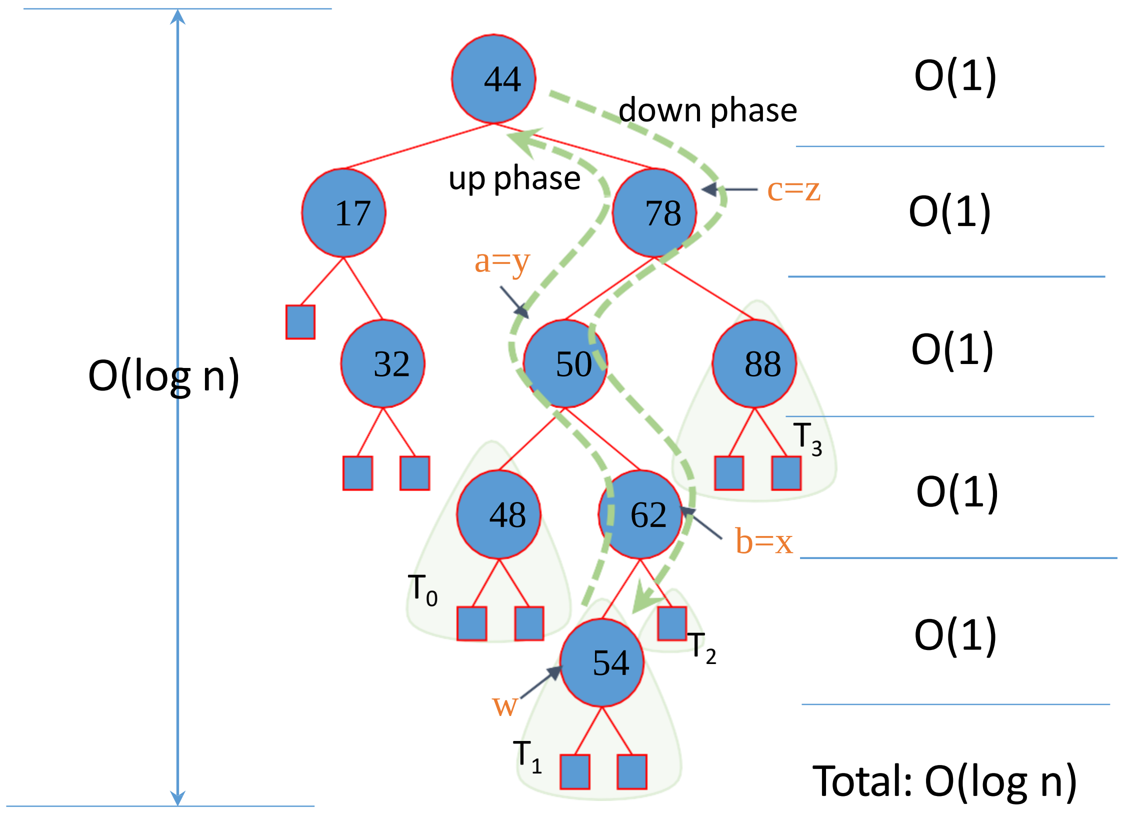
- Deletion in an AVL tree is the same as in a BST
- But balance has to be restored
- After the deletion:
  - Let w be the parent of the node being removed.
  - Let z be the first unbalanced node encountered while travelling up the tree from w
  - Also, let y be the child of z with the largest height, and let x be the child of y with the largest height
  - We perform restructure(x) to restore balance at z
  - As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



AVL trees - performance

Worst case:

- a single restructure is O(1)
  - using a linked-structure binary tree
- Search or find is O(log n)
  - height of tree is O(log n), no restructuring needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- remove is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

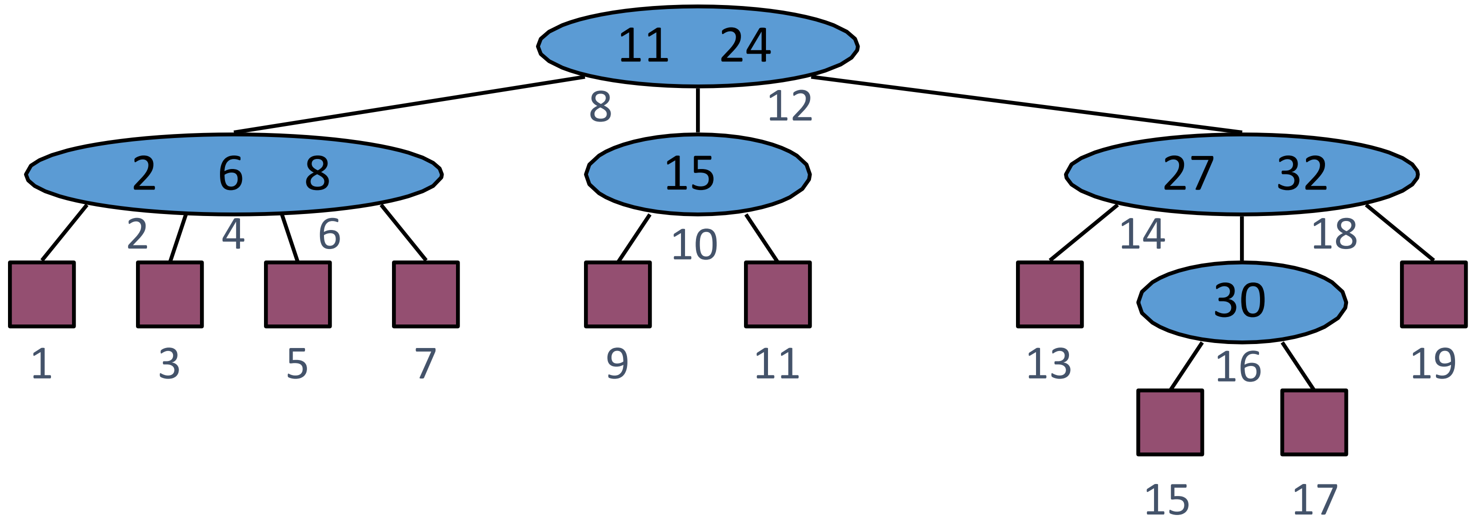


Multi-way search trees

- A multi-way search tree is an ordered tree such that
  - Each internal node has at least two children and stores $d - 1$ key-element items (k_i, o_i), where d is the number of children
  - For a node with children $v_1\ v_2\ \dots\ v_d$ storing keys $k_1\ k_2\ \dots\ k_{d-1}$
    - keys in the subtree of v_1 are less than k_1
    - keys in the subtree of v_i are between k_{i-1} and k_i $(i = 2, \dots, d-1)$
    - keys in the subtree of v_d are greater than k_{d-1}
- The height of a multi-way search tree is O(log n)

Inorder traversal:

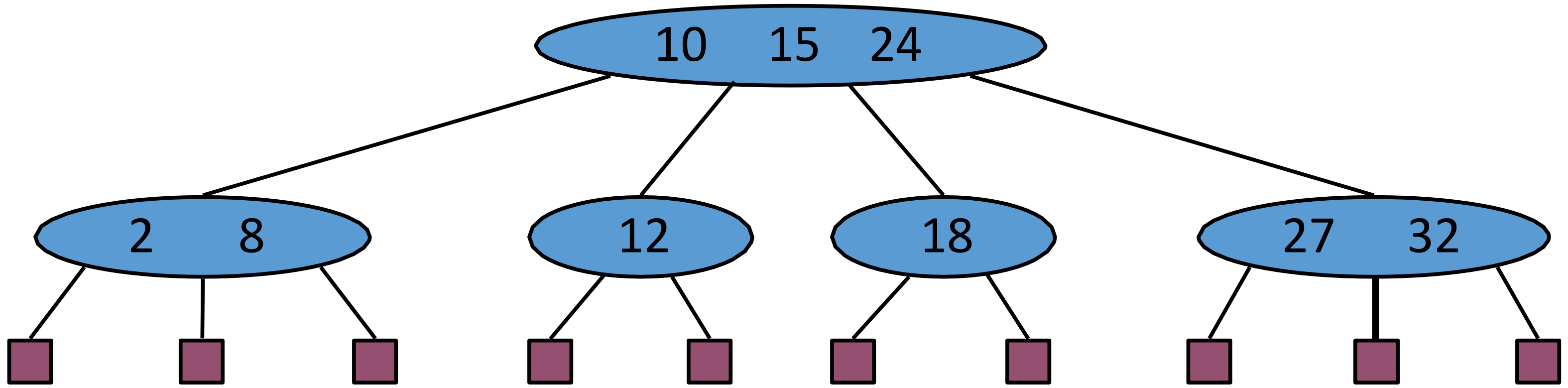
- (k_i, o_i) of node v is visited between the recursive traversals of the subtrees of v rooted at children v_i and v_{i+1}
- The keys of the multi-way search tree are visited in increasing order



(2,4) trees

- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search tree with the following properties:
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node

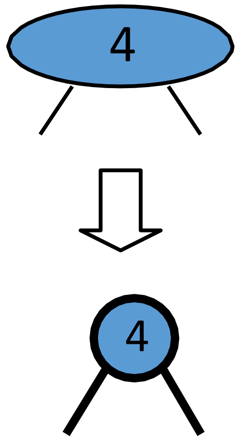
- Theorem: A (2,4) tree storing n items has height $O(\log n)$
- Searching in a (2,4) tree with n items takes $O(\log n)$ time

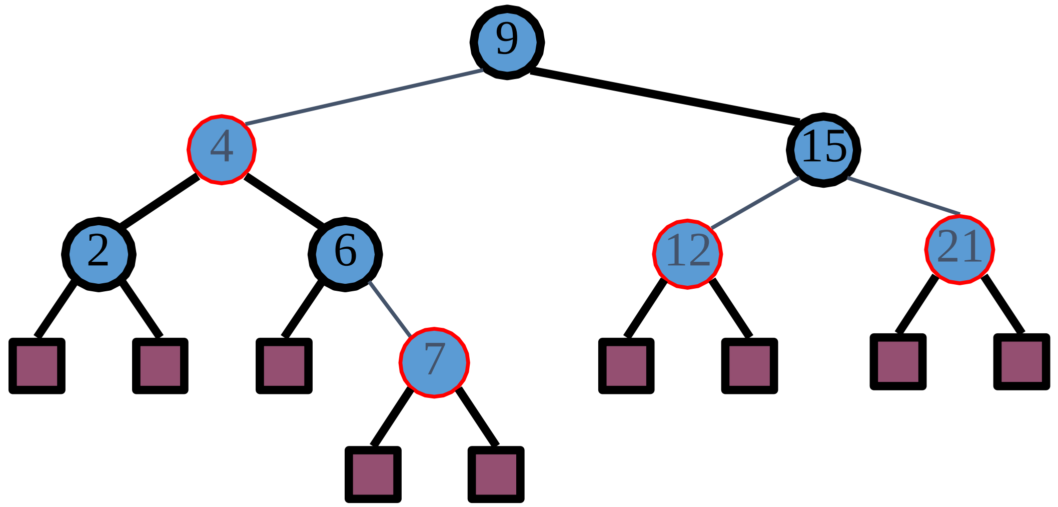


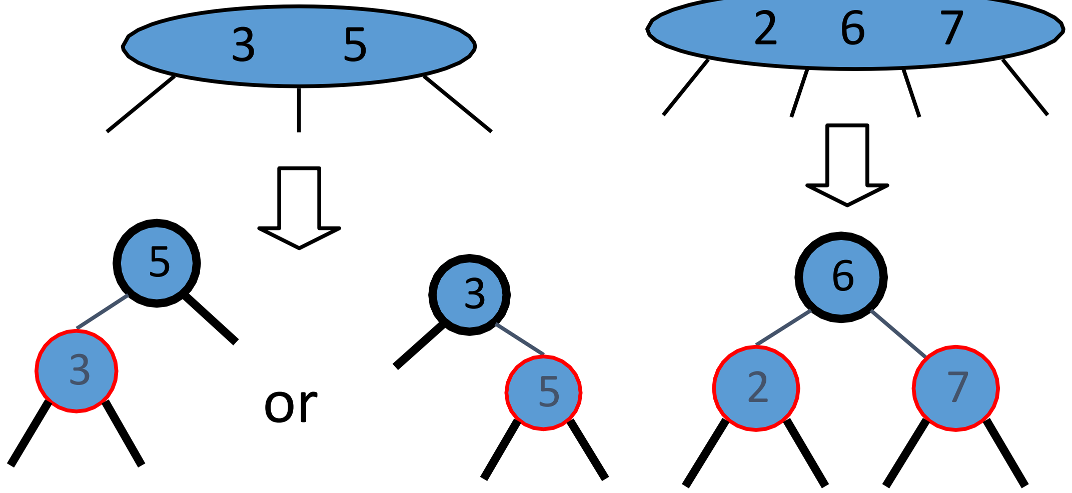
Red-black trees

- A red-black tree is a BST that satisfies the following properties:
  - Root Property: the root is black
  - External Property: every leaf is black
  - Internal Property: the children of a red node are black
  - Depth Property: all the leaves have the same black depth

- A (2,4) tree can be easily converted into a red-black tree
- The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is $O(\log n)$
- Then, the height of a red-black tree is O(log n)



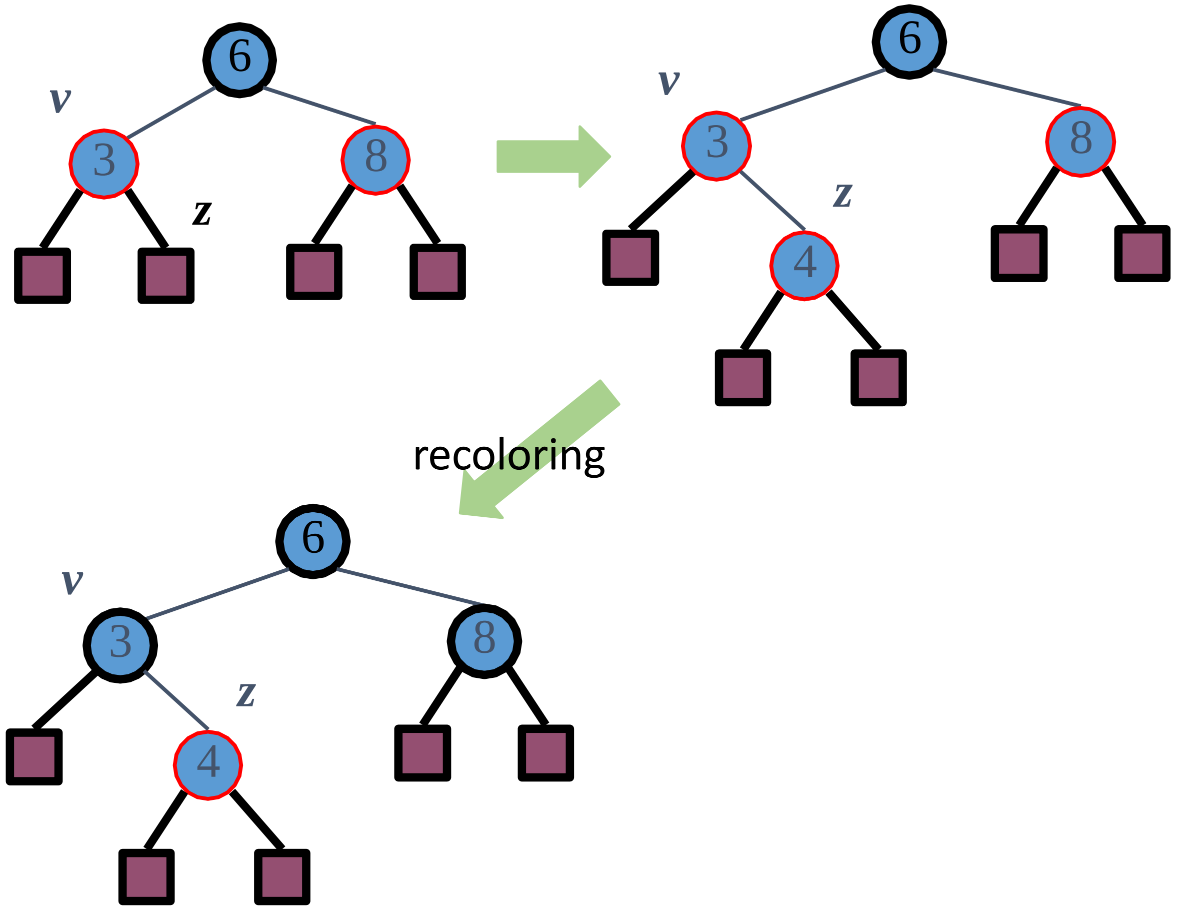




Red-black trees - insertion

- Insertions in a red black tree is like in a BST, where the newly inserted node z is colored red, unless it is the root
  - We preserve the root, external, and depth properties
  - If the parent v of z is black, we also preserve the internal property and we are done
  - Else (v is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree
- Reorganization requires:
  - Restructuring
  - Recoloring

Example: insert 4



Red-black trees – deletion

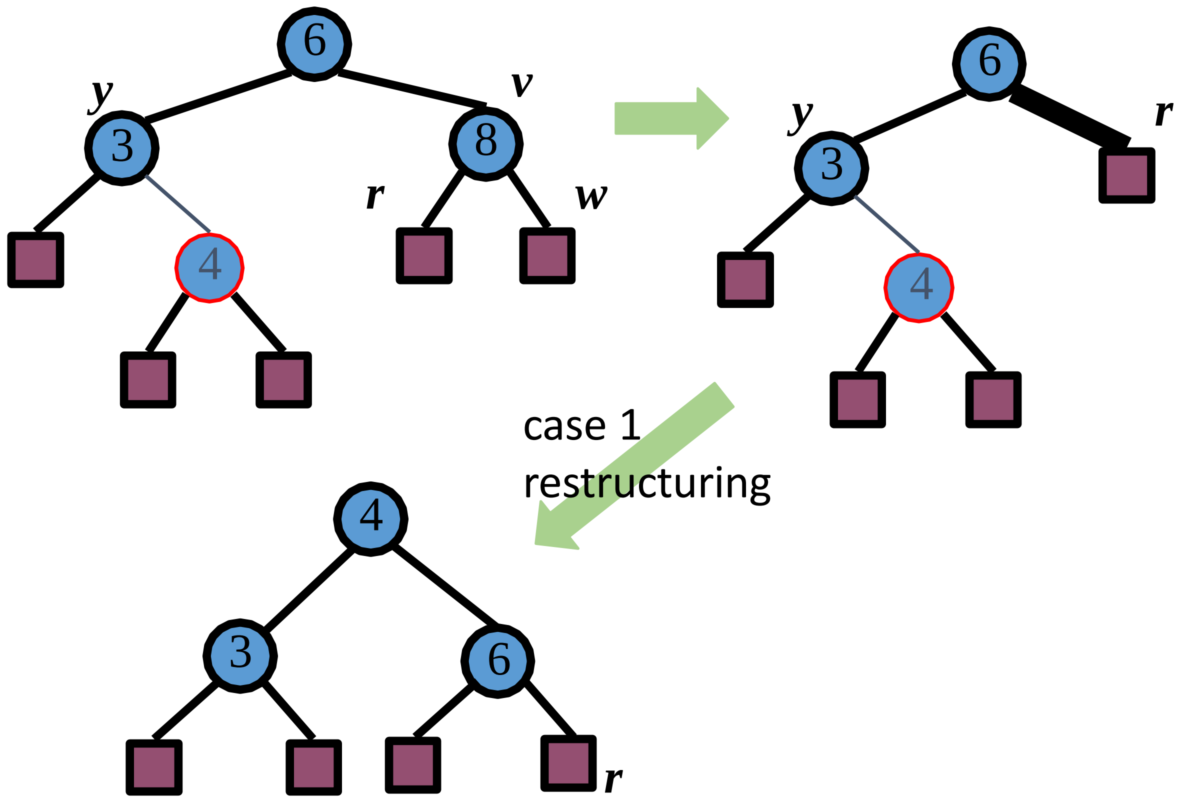
- To perform operation remove(k), we first execute the deletion algorithm for BST
- Let v be the internal node removed, w the external node removed, and r the sibling of w
  - If either v or r was red, we color r black and we are done
  - Else (v and r were both black) we color r ***double black***, which is a violation of the depth property requiring restructuring the tree
- Let x be the parent of r, and y be the sibling of r
- The algorithm for remedying a double black node r with sibling y considers three cases:

  Case 1: y is black and has a red child z
  - We perform a restructuring , and we are done

  Case 2: y is black and its children are both black
  - We perform a recoloring, which may propagate up the double black violation

  Case 3: y is red (and hence it has a black child z)
  - We perform an adjustment, after which either Case 1 or Case 2 applies

Example: remove 8

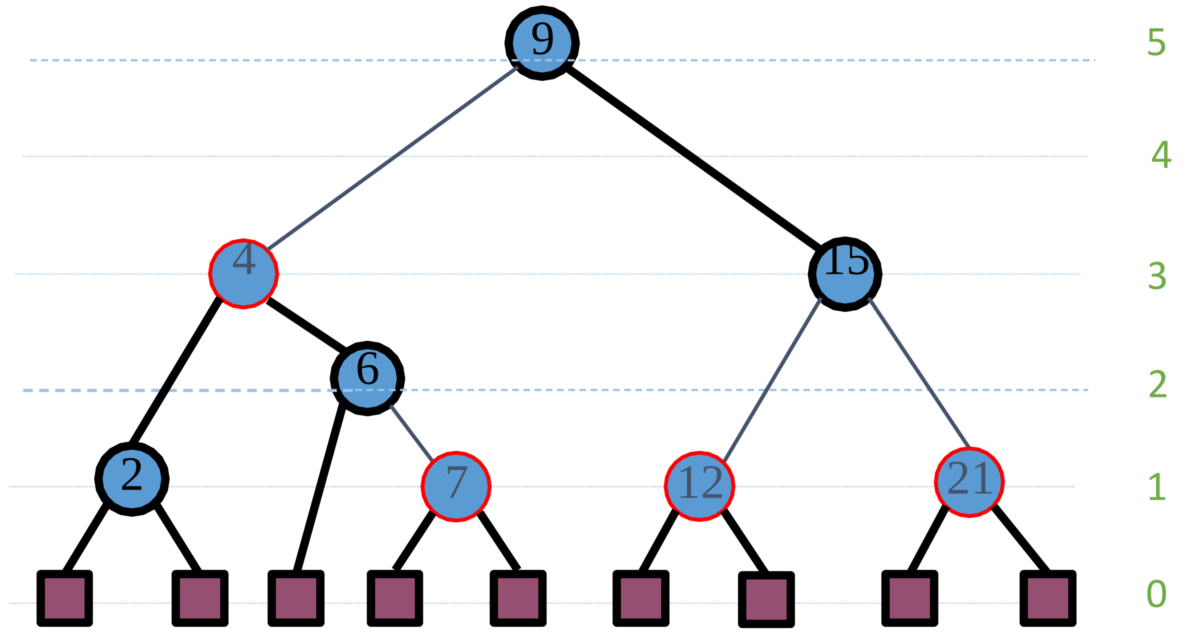


Worst-case analysis:

- Insertion and deletion in a red-black tree take $O(\log n)$ time.
- Search or find also takes $O(\log n)$ time.

Weak AVL trees (optional – not in exam)

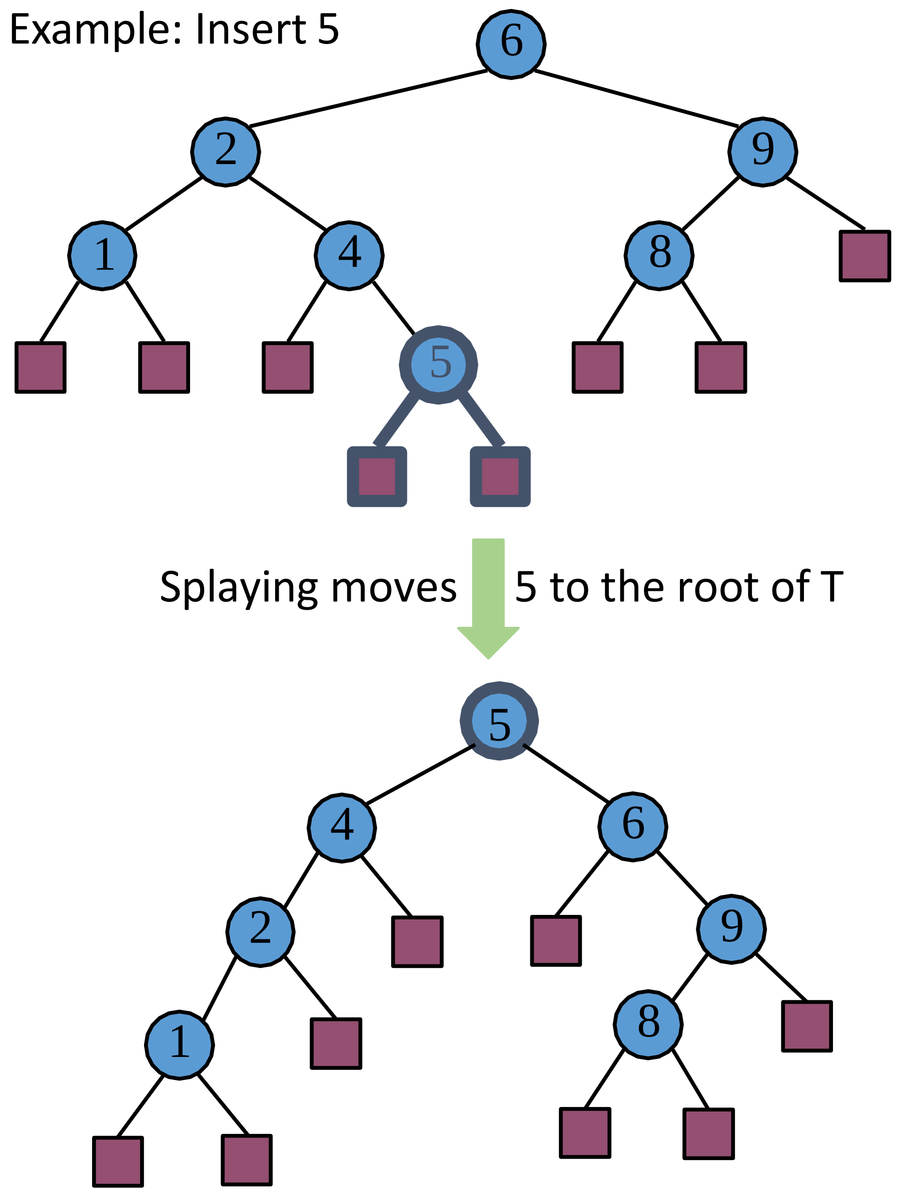
- A weak AVL tree or wavl tree has features of both AVL and red-black trees
- Insertions and deletions are simpler though
- Definition:
  - Every node v has a rank, r(v)
  - For every node v, r(v) < r(parent(v))
- Rank difference of v: difference between r(v) and r(parent(v))
- Properties of rank:
  - The rank difference of any non-root node is 1 or 2
  - Every external node has rank 0
  - Leaf children of an internal node with two leaves as children, the children must not have a rank difference of 2
- Main properties of wavl trees:
  - Every AVL tree is a wavl tree
  - Every wavl tree can be colored as a red-black tree
  - Every red-black tree can be made a wavl tree after rank assignments



- The height of a wavl tree storing n keys is at most 2 log(n + 1)
- Insertion: Similar to AVL trees, but rebalancing using *promotion*.
- Deletion: Similar to AVL trees, but rebalancing using *demotion*.
- Search, insertion and deletion in a wavl tree take O(log n)
- More details found in Section 4.4 of [1].

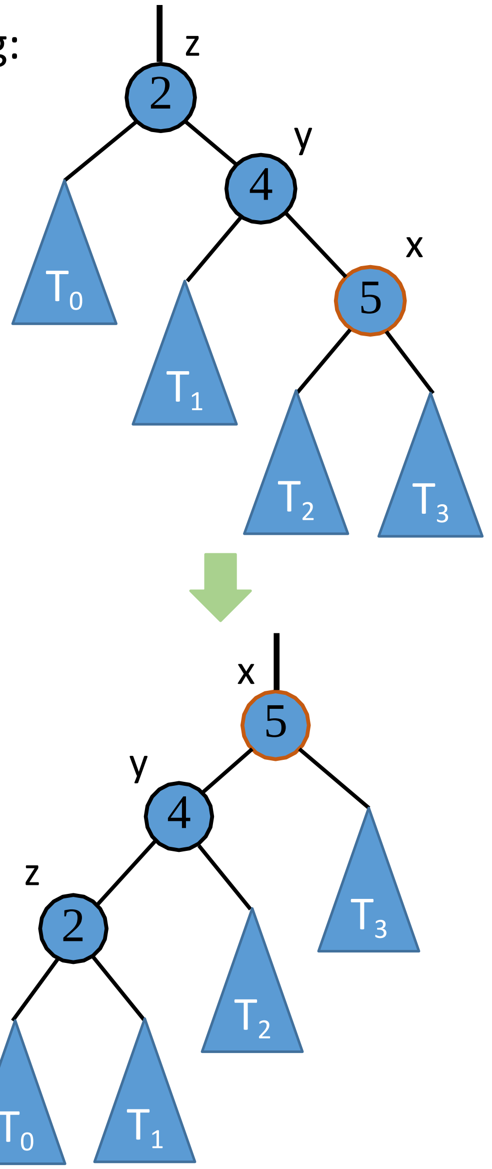
Splay trees

- Splay trees are BSTs
- Unlike other trees, they don't store any height, balance, colors, etc.
- Splaying is the main operation:
  - Bottommost position p after search, insertion or deletion is moved up to the root
  - More frequently accessed nodes stay closer to the root
- Most important feature: average-case running time of search, insertion and deletion is O(log n)
- Worst-case running time of search, insertion and deletion is like in BSTs: O(n)
- Three main splaying operations:
  - Zig-zig
  - Zig-zag
  - Zig

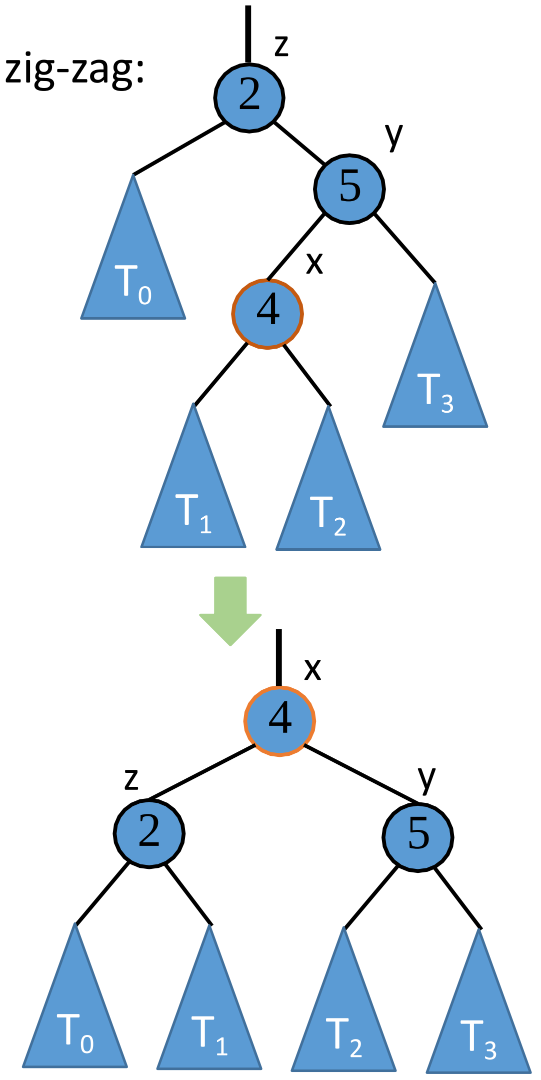


Splaying

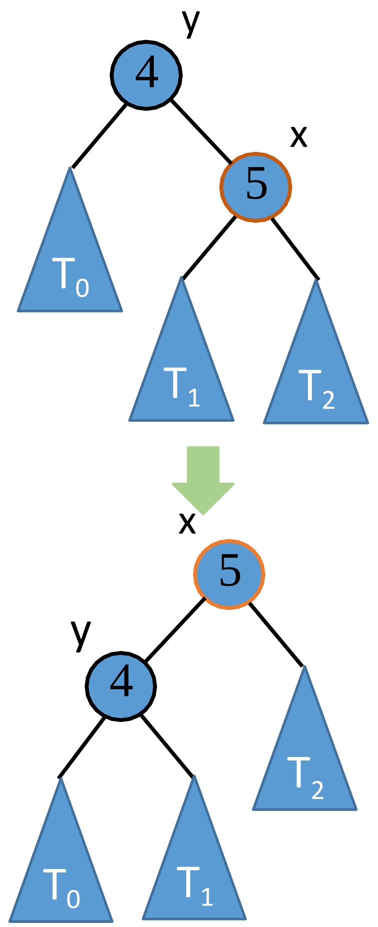
zig-zig:



zig-zag:



zig:

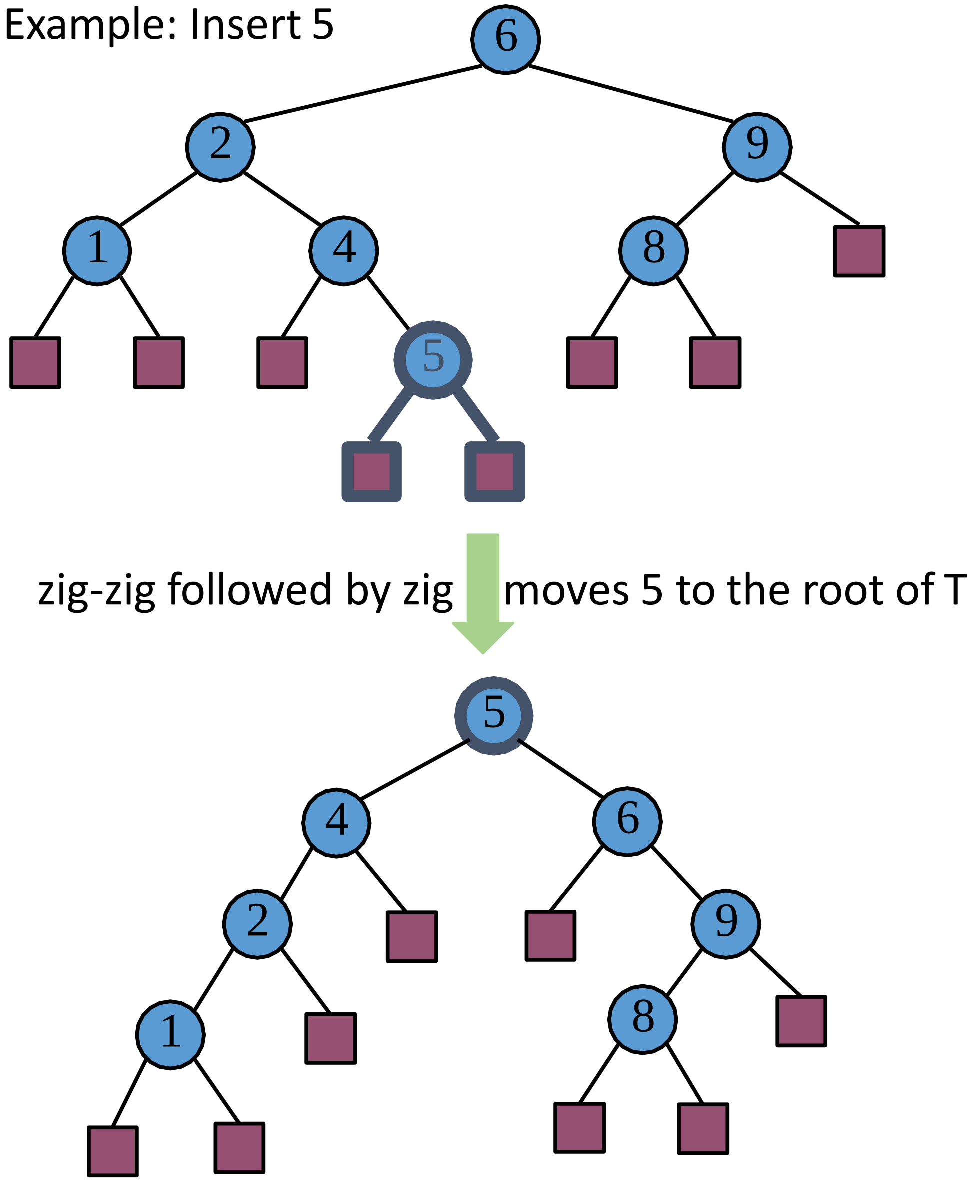


x doesn't have any grandparent

Splaying - when to splay

- Search:
  - When searching for a key k, which is found at node x
  - x is splayed
- Insertion:
  - When inserting a key k, a new internal node x is created, where x contains k
  - x is splayed
- Deletion:
  - When deleting a key k that is contained in node w, the parent x of w is splayed
- Performance
  - Search, insertion and deletion take O(log n) average-case running time
  - Worst-case is still O(n)
  - In practice, splay trees perform exceptionally well

Example: Insert 5



Review and comparison

Search tree	Average case			Worst case		
	Search	Insertion	Deletion	Search	Insertion	Deletion
BST	O(log n)*	O(log n)*	O(log n)*	O(n)	O(n)	O(n)
AVL	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Red-black	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
wavl	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Splay	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)

* Average case for a BST assumes a series of insertions and deletions of n randomly generated keys.

Review and Further Reading

- TreeSet and TreeMap classes in Java 8 implement balanced search trees
- They implement and maintain a red-black tree.
- TreeSet implementationis based on TreeMap.
- TreeSet
  - http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html
- TreeMap
  - http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html
- BST, AVL, splay and red-black trees:
  - Ch. 11 of [2], Ch. 4 of [1]
  - Ch. 4 of [3]
- Implementations in Standard library
  - Sec. 4.8 of [3]

References

1. Algorithm Design and Applications by M. Goodrich and R. Tamassia, Wiley, 2015.
2. Data Structures and Algorithms in Java, 6th Edition, by M. Goodrich and R. Tamassia, Wiley, 2014.
3. Data Structures and Algorithm Analysis in Java, 3rd Edition, by M. Weiss, Addison-Wesley, 2012.
4. Java documentation:
   - http://docs.oracle.com/javase/8/docs/api/overview-summary.html
   - http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html
   - http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

Lab - Practice

Use classes BinarySearchTree, AVLTree, RedBlackBST, SplayTree to create four search trees.

1. For each tree:
   a) Insert 100,000 integer keys, from 1 to 100,000 (ascending order). Find the average time of each insertion.
   b) Do 100,000 searches of random integer keys between 1 and 100,000. Find the average time of each search.
   c) Delete all the keys in the trees, starting from 100,000 down to 1 (descending order). Find the average time of each deletion.
2. For each tree:
   a) Insert 100,000 random keys between 1 and 100,000. Find the average time of each search.
   b) Repeat #1.b.
   c) Repeat #1.c, but with random keys between 1 and 100,000. Note that not all the keys can be found in the tree.

Exercises

1. Define height, depth, internal/external node, subtree. Give examples.
2. Define a binary tree. Give an example of a decision tree.
3. Give examples for the properties of binary trees. Show the facts.
4. Write recursive algorithms for printing a binary tree and for printing an arithmetic expression.
5. *Write a program that reads all the folders and files and stores them in a tree (not necessarily binary). List all the folders and files along with their size.
6. Write an algorithm that modifies binary search by finding the keys in a certain range. Show that the algorithm runs in O(log n).
7. Give an example of binary tree that is not proper.
8. Give a sequence of keys that after being inserted in an empty BST, it produces a: (a) balanced tree, (b) completely unbalanced tree.
9. Give examples of AVL trees with the minimum number of keys, and whose heights are 4, 5, 6, 7.
10. Consider the tree of slide 13. Insert 56 and rebalance the tree. Delete 44 and rebalance the tree.
11. Write algorithms for insertion and deletion in an AVL tree.
12. Write an algorithm that converts a (2,4) tree into a red-black tree
13. Show that the heights of a red-black tree and the (2,4) tree are the same: O(log n).
14. Write algorithms for insertion and deletion in a red-black tree.
15. Give an example of a wavl tree with 8 nodes, and show how that tree can be converted into an AVL tree and a red-black tree.
16. Show (no formal proof required) that a wavl tree has height O(log n).
17. Investigate in ref [1] how insertions and deletions are performed in a wavl tree.
18. Write an algorithm for insertion in a splay tree.
19. Consider the red-black tree of slide 17. Insert 5 and delete 8.
20. Give an example of keys that inserted into a splay tree, it results in a balanced tree (like an AVL tree).
21. Consider the tree of slide 21. Insert 1, and then 1. Apply splaying after each insertion.
22. *Write a program that read the words of a file and insert them in a search tree (BST, AVL, etc.). Also use the hash table of previous chapter. Do 1000 searches and compare the average search times.
23. *Write an algorithm that given two AVL trees and a key, it merges them into a single AVL tree. Do the same for a BST. Can you optimize the algorithm?
24. Write an algorithm that converts an AVL tree into a red-black tree. Can you optimize your algorithm?