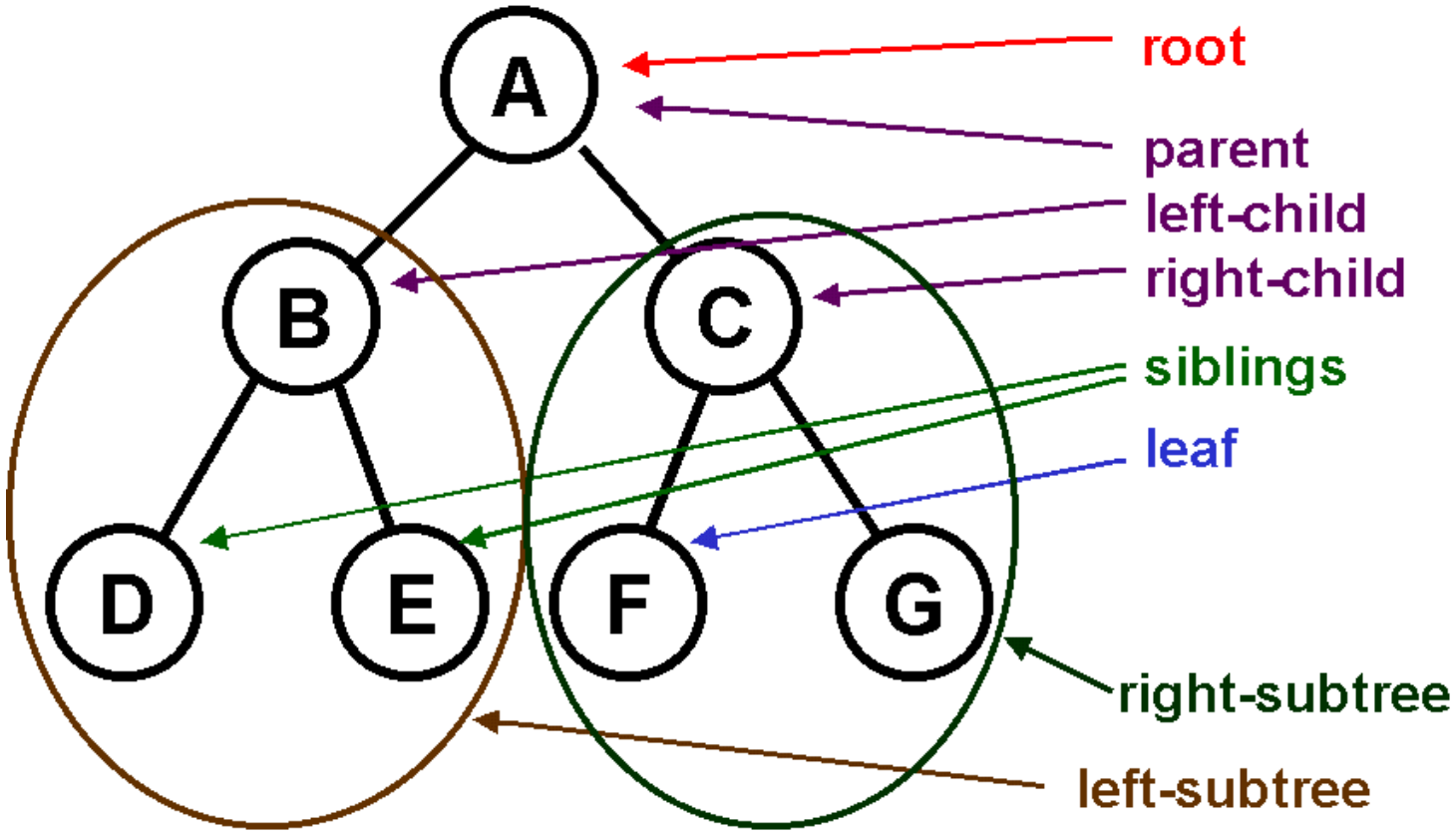
trees

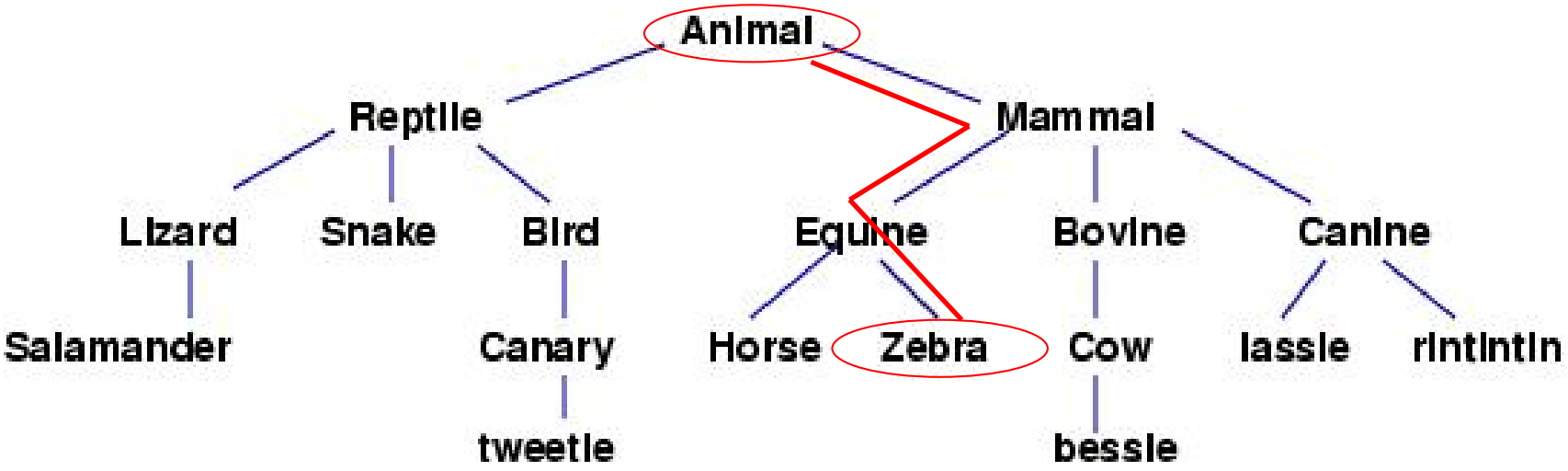
what's a tree?

- Tree is defined **recursively**
- A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node r, called the **root**, and zero or more non-empty (sub) trees T_1, T_2, …, T_k each of whose roots are connected by a directed edge from r.
- A tree is a collection of N nodes, one of which is the root and N-1 edges.
- The root of each subtree is said to be a **child** of r and r is said to be the **parent** of each subtree root.
- **Leaves**: nodes with no children (also known as external nodes)
- **Internal Nodes**: nodes with children
- **Siblings**: nodes with the same parent


A
B
C
D
E
F
G
root
parent
left-child
right-child
siblings
leaf
right-subtree
left-subtree

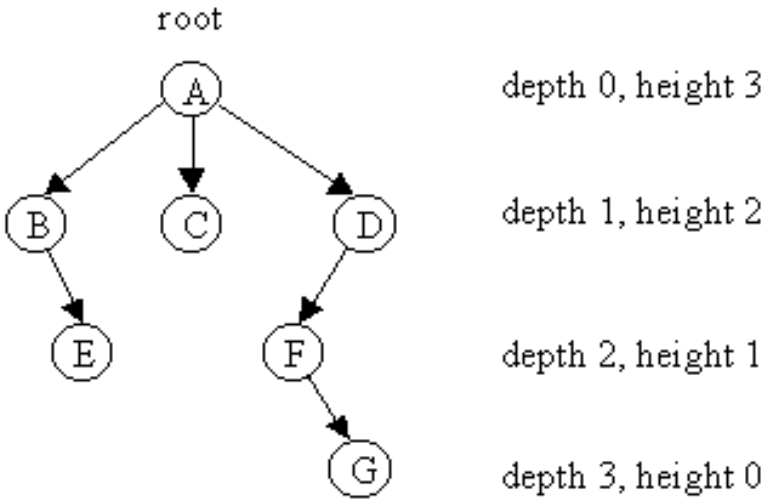
path

- A **path** from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, …, n_k such that n_i is the parent of n_{i+1} for 1<= i <= k.
- The length of this path is the number of edges on the path namely k-1.
- The length of the path from a node to itself is 0.
- There is exactly one path from from the root to each node.



depth-height

- Depth (of node): the length of the unique path from the root to a node.
- Depth (of tree): The depth of a tree is equal to the depth of its deepest leaf.
- Height (of node): the length of the longest path from a node to a leaf.
  - All leaves have a height of 0
  - The height of the root is equal to the depth of the tree



A tree of height 3

Binary tree

- A binary tree is a tree in which no node can have more than two children.
- Each node has an element, a reference to a left child and a reference to a right child
- A data structure that supports dynamic set operations:
  - Insert
  - Delete
  - Search
  - Predecessor
  - Successor
  - Minimum
  - Maximum
- It can be used both as a dictionary and as a priority queue
- All the above operations are O(h), with h being the height of the tree
- Average case: O(*lg*n); worst case: O(n)

traversals

- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once
- Tree traversals are naturally recursive
- Since a binary tree has three “parts,” there are six possible ways to traverse the binary tree:
  - root, left, right
  - left, root, right
  - left, right, root
  - root, right, left
  - right, root, left
  - right, left, root

preorder traversal

- In preorder, the root is visited *first*
- Here's a preorder traversal to print out all the elements in the binary tree:

```
public void preorderPrint(BinaryTree bt) {
     if (bt == null) return;
     System.out.println(bt.value);
     preorderPrint(bt.leftChild);
     preorderPrint(bt.rightChild);
}
```

inorder traversal

- In inorder, the root is visited *in the middle*
- Here's an inorder traversal to print out all the elements in the binary tree:

```
public void inorderPrint(BinaryTree bt) {
     if (bt == null) return;
     inorderPrint(bt.leftChild);
     System.out.println(bt.value);
     inorderPrint(bt.rightChild);
}
```

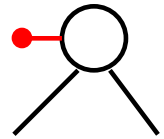
postorder traversal

- In postorder, the root is visited *last*
- Here's a postorder traversal to print out all the elements in the binary tree:

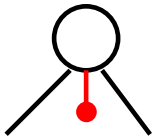
```
public void postorderPrint(BinaryTree bt) {
     if (bt == null) return;
     postorderPrint(bt.leftChild);
     postorderPrint(bt.rightChild);
     System.out.println(bt.value);
}
```

traverse a tree: simple schema

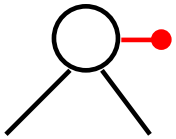
- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:



preorder



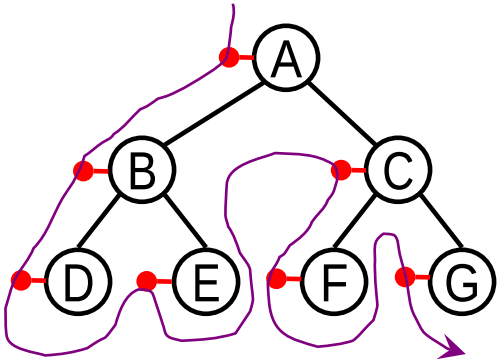
inorder



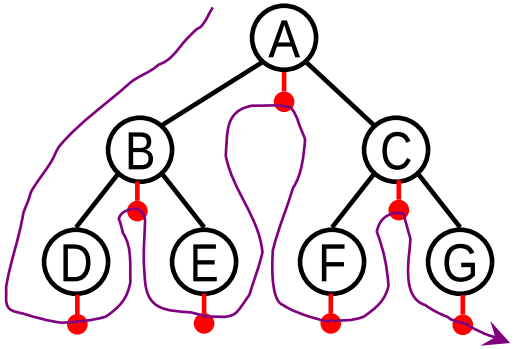
postorder

preorder: root-left-right **inorder: left-root-right** **postorder: left-right-root**

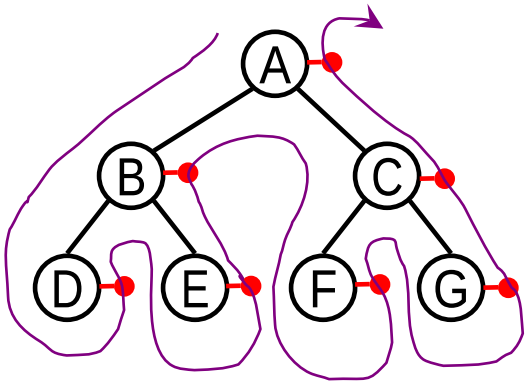
- To traverse the tree, collect the flags:



A B D E C F G



D B E A F C G



D E B F G C A

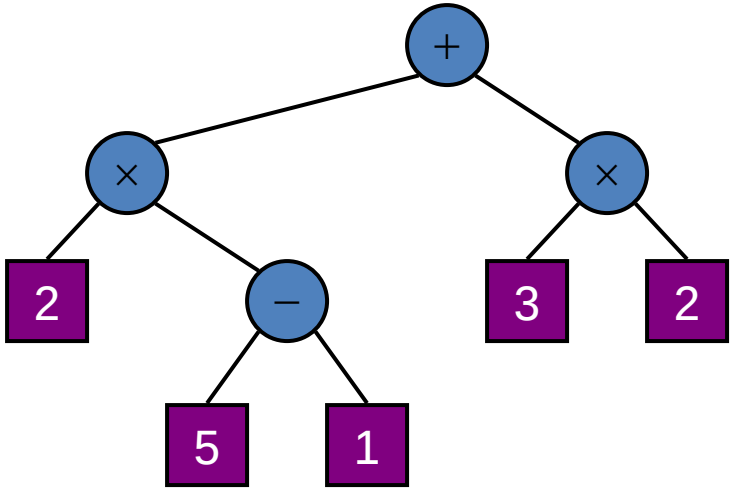
other traversals

- The other traversals are the reverse of these three standard ones
  - Reverse preorder: root, right subtree, left subtree
  - Reverse inorder: right subtree, root, left subtree
  - Reverse postorder: right subtree, left subtree, root

example: arithmetic expression tree

- Binary tree for an arithmetic expression
  - internal nodes: operators
  - leaves: operands
- Example: arithmetic expression tree for the expression

$$((2 \times (5 - 1)) + (3 \times 2))$$

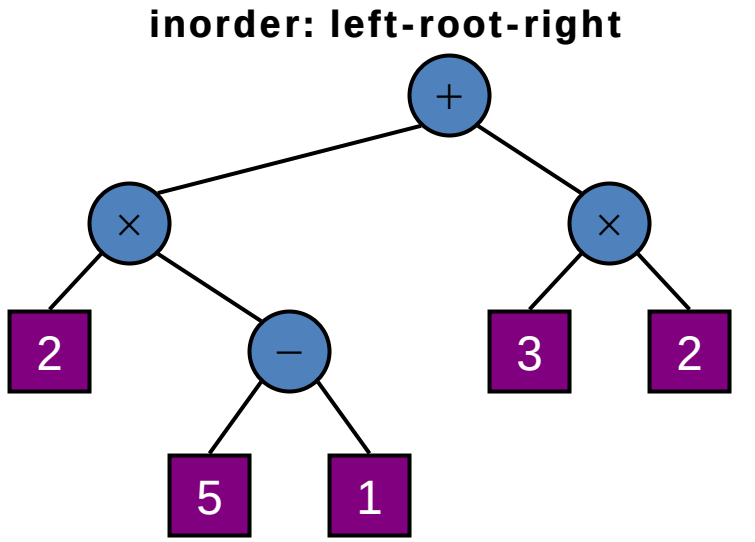


inorder: left-root-right

print

- inorder traversal:
  - print “(“ before traversing left subtree
  - print operand or operator when visiting node
  - print “)“ after traversing right subtree

```
void printTree(t)
    if (t.left != null)
        print("(");
        printTree (t.left);
    print(t.element );
    if (t.right != null)
        printTree (t.right);
        print (")");
```

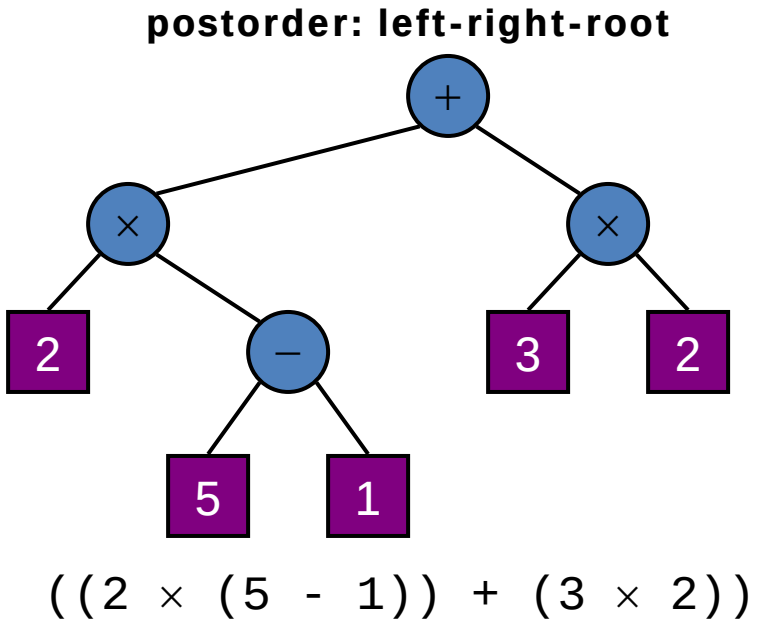


((2 × (5 - 1)) + (3 × 2))

evaluate

- postorder traversal
  - recursively evaluate subtrees
  - apply the operator after subtrees are evaluated

```
int evaluate (t)
    if (t.left == null)
      //external node
      return t.element;
    else  //internal node
      x = evaluate (t.left);
      y = evaluate (t.right);
      let o be the operator
        t.element
      z = apply o to x and y
      return z;
```

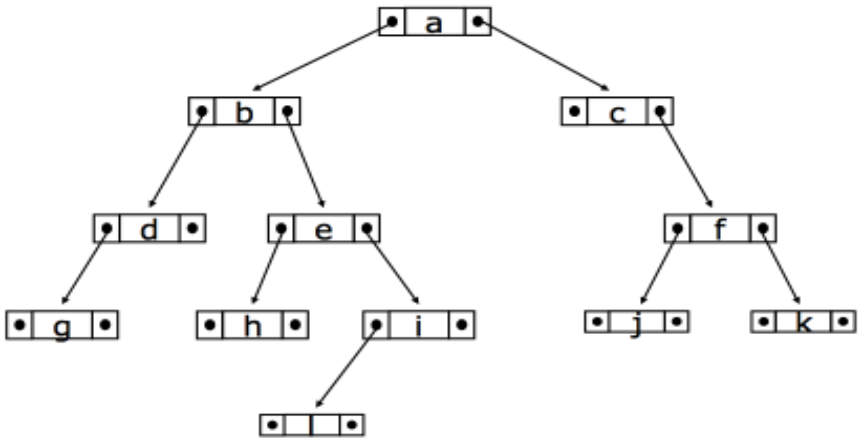


BST

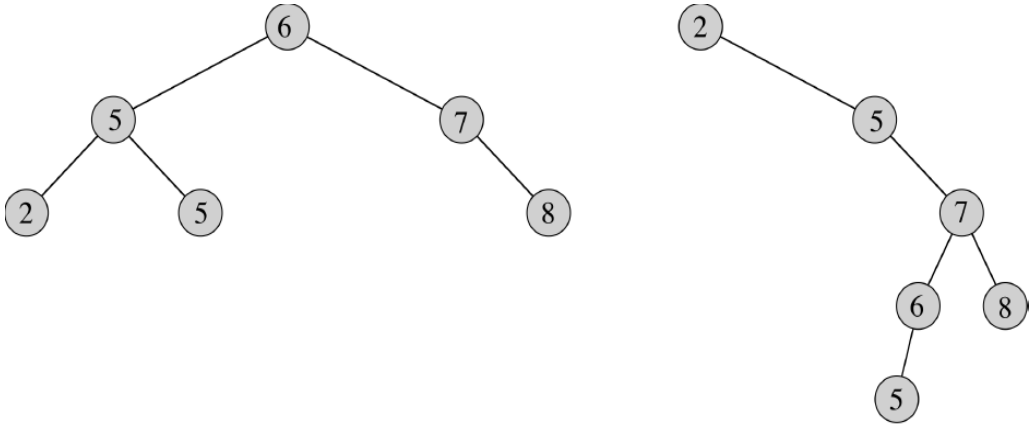
- Binary tree
- Each node holds an object (key plus “satellite” data)
- Each node has a left, right, and parent pointer
- Root: parent = NIL; Leaf: left/right child = NIL
- The keys are stored in a way that satisfies the BST property:

If x is a node in a BST and y is in the left subtree of x, then y.key ≤ x.key. If y is in the right subtree of x, then y.key > x.key

- Using BST we can print all the keys in sorted order by **in-order tree walk**



in-order walk

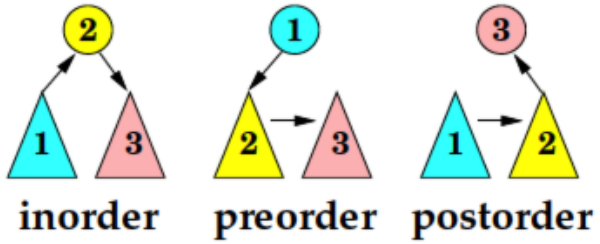



```
Inorder-Tree-Walk (x)
  if x != NIL
    Inorder-Tree-Walk(x.left)
    print x.key
    Inorder-Tree-Walk(x.right)
```

If x is the root of an n-node subtree, then the call Inorder-Tree-Walk(x) takes Θ(n) time

traversal of the nodes

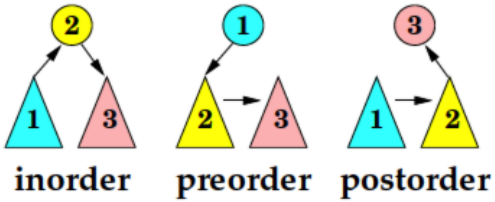
- By “traversal" we mean visiting all the nodes in a graph.
- Traversal strategies can be specified by the ordering of the three objects to visit: the current node, the left subtree, and the right subtree.
- We assume the the left subtree always comes before the right subtree.



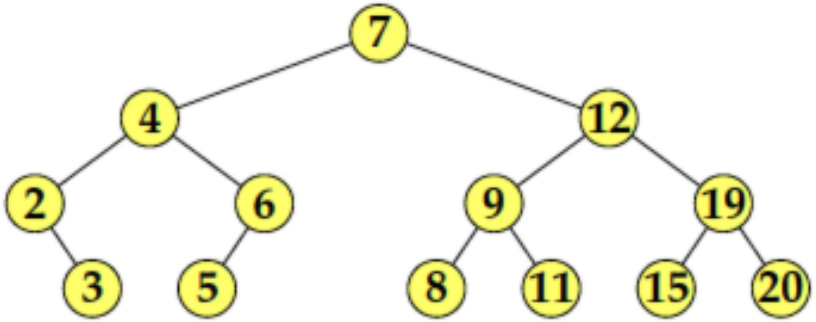
1. **Inorder.** The ordering is: the left subtree, the current node, the right subtree.

2. **Preorder.** The ordering is: the current node, the left subtree, the right subtree.

3. **Postorder.** The ordering is: the left subtree, the right subtree, the current node.



example



- Inorder traversal:

  2, 3, 4, 5, 6, 7, 8 , 9, 11, 12, 15, 19, 20

- Preorder traversal:

  7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20

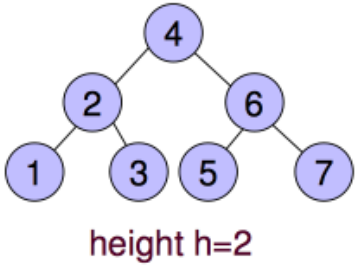
- Postorder traversal:

  3, 2, 5, 6, 4, 8, 11, 9, 15, 20, 19, 12, 7

building BST

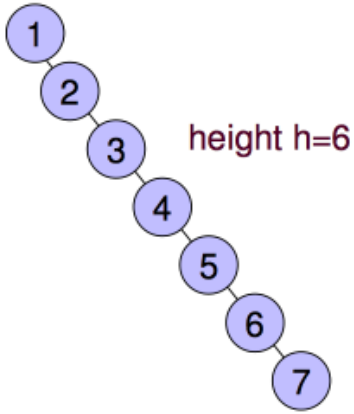
- Using just the BST property, we can build a tree knowing the order in which the values are inserted.
- The order affects the resulting tree
  <u>example:</u> build the 2 trees when the values are inserted in order.

4, 2, 3, 6, 5, 7, 1



height h=2

1, 2, 3, 4, 5, 6, 7



height h=6

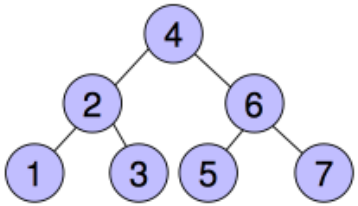
inorder: left-root-right

- If the values are inserted in order, we essentially get a linked list.
- We would say that the tree at left is more balanced

balance

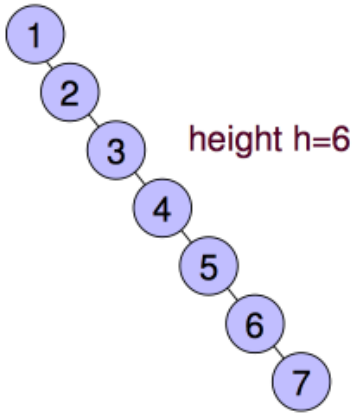
- Most BST algorithms depend on the height h.
- So, the smaller the height of the tree, the faster the algorithms will be.
- To get a small height, we want our tree to be as balanced as possible.
- So the values should be inserted in random order.
- Searching the left tree is just like O(logN) binary search.
- Searching the right tree is O(N) linear search.

4, 2, 3, 6, 5, 7, 1



height h=2

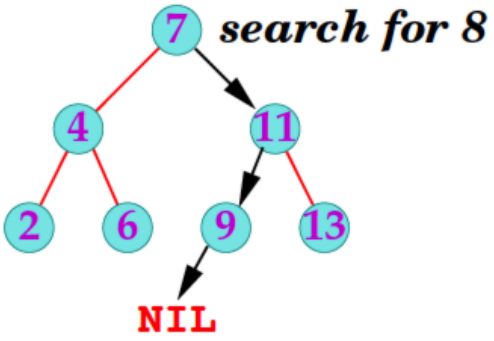
1, 2, 3, 4, 5, 6, 7



height h=6

searching

```
BST-Search(x, k)
 1: y ← x
 2: while y ≠ nil do
 3:     if key[y] = k then return y
 4:     else if key[y] < k then y ← right[y]
 5:     else y ← left[y]
 6: return ("NOT FOUND")
```



- We assume that a key and the subtree in which the key is searched for are given as an input
- Suppose we are at a node. If the node has the key that is being searched for, then the search is over.
- Otherwise, the key at the current node is either strictly smaller than the key that is searched for or strictly greater than the key that is searched for.
- In the first case, by the BST property, all the keys in the left subtree are strictly less than the key that is searched for. That means that we do not need to search in the left subtree. Thus, we will examine only the right subtree.
- In the second case, by symmetry we will examine only the left subtree.

max and min

- To find the minimum identify the leftmost node, i.e. the farthest node you can reach by following only left branches.

```
BST-Minimum(x)
 1: if x = nil then return ("Empty Tree")
 2: y ← x
 3: while left[y] ≠ nil do y ← left[y]
 4: return (key[y])
```

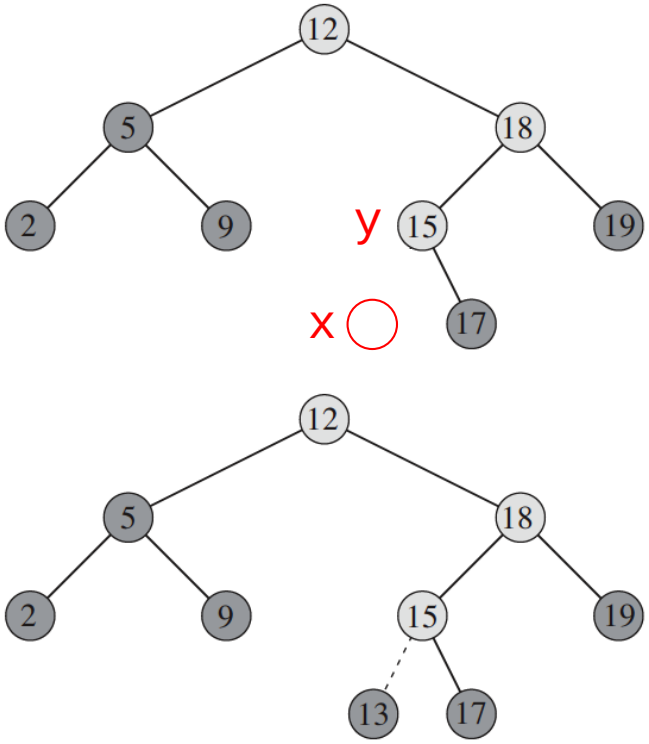
- To find the maximum identify the rightmost node, i.e. the farthest node you can reach by following only right branches.

```
BST-Maximum(x)
 1: if x = nil then return ("Empty Tree")
 2: y ← x
 3: while right[y] ≠ nil do y ← right[y]
 4: return (key[y])
```

insertion

- Suppose that we need to insert a node z such that k = key[z]. Using binary search we find a nil such that replacing it by z does not break the BST-property.

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z          // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```

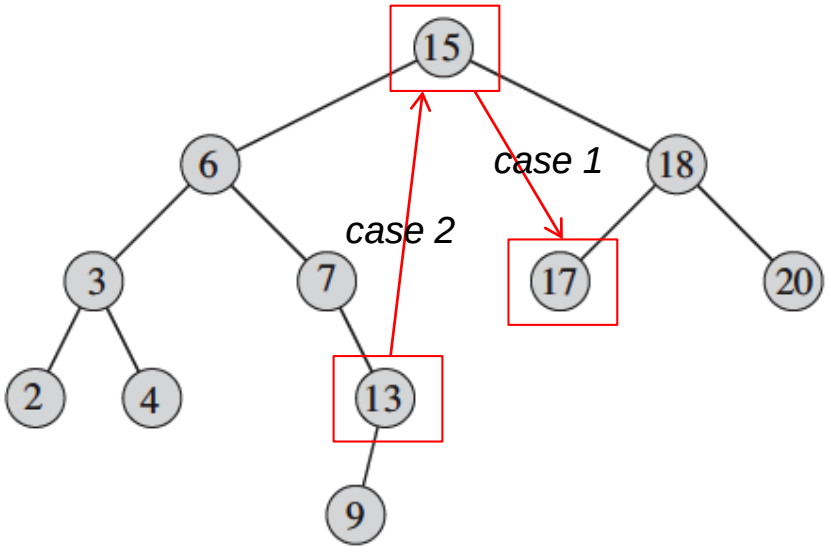


successor

The successor of a key k in a search tree is the smallest key that belongs to the tree and that is strictly greater than k.

The idea for finding the successor of a given node x:

- If x has the right child, then the successor is the minimum in the right subtree of x.
- Otherwise, the successor is the parent of the farthest node that can be reached from x by following only right branches backward.

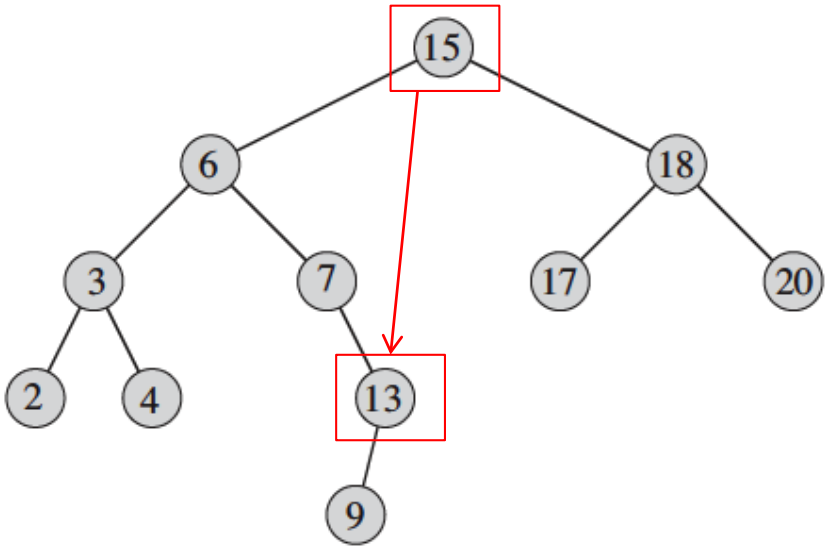


predecessor

The predecessor of a key k in a search tree is the largest key that belongs to the tree and that is strictly less than k.

The idea for finding the predecessor of a given node x:

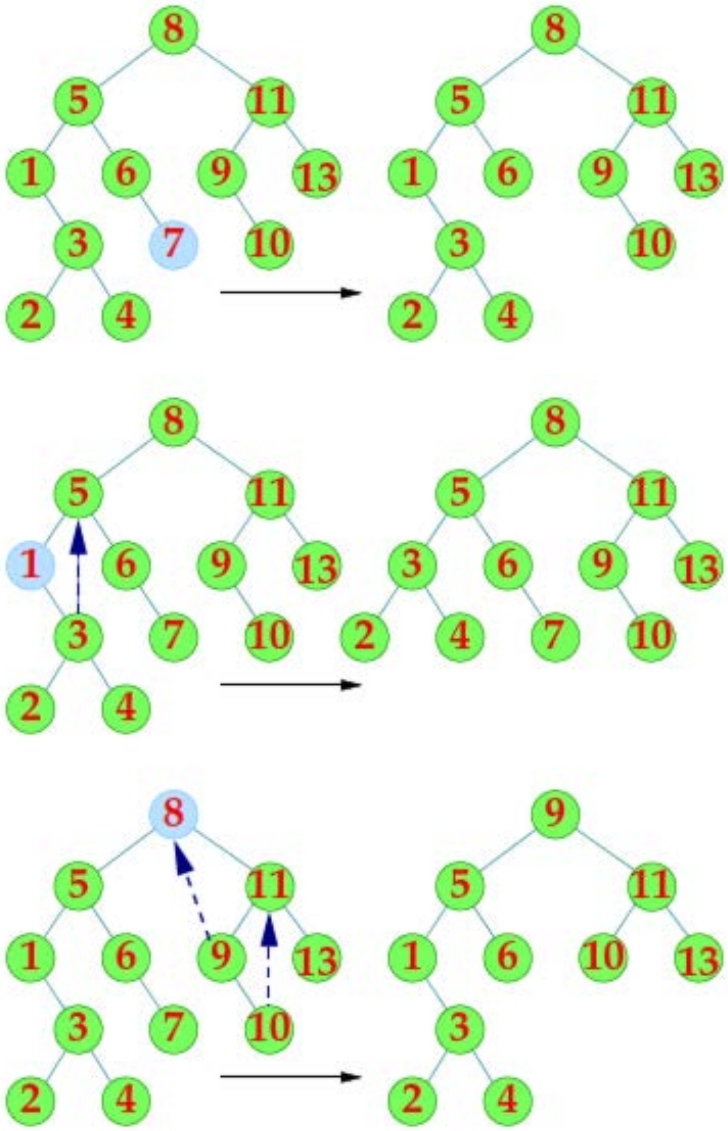
- If x has the left child, then the predecessor is the maximum in the left subtree of x.
- Otherwise, the predecessor is the parent of the farthest node that can be reached from x by following only left branches backward.



deletion

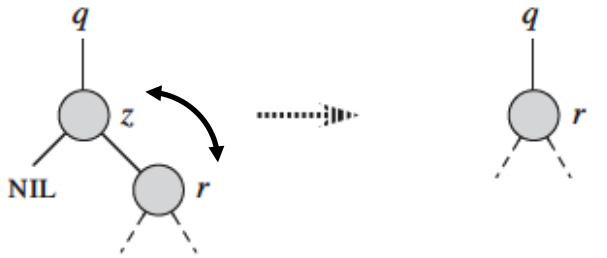
Suppose we want to delete a node **z**.

1. If **z** has no children, then we will just replace **z** by nil.
2. If **z** has only one child, then we will promote the unique child to **z**'s place.
3. If **z** has two children, then we will identify **z**'s successor **y** and…

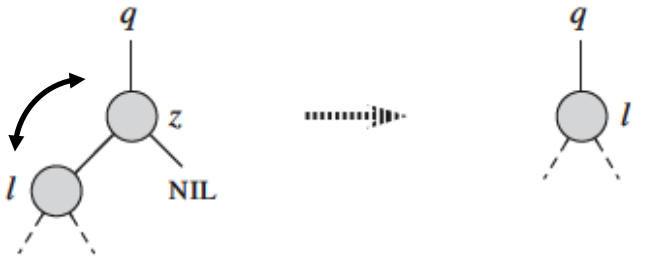


deleting z

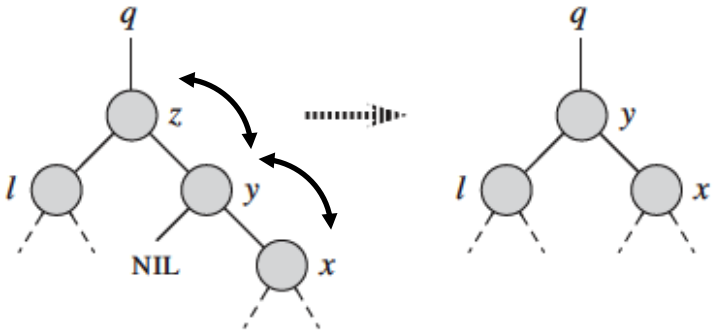
No left child



No right child



Both children,
y=successor(z)




```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

TRANSPLANT(T,***u***,***v***) replaces the subtree rooted at node ***u*** with the subtree rooted at node ***v***, node ***u***'s parent becomes node ***v***'s parent, and ***u***'s parent ends up having ***v*** as its appropriate child.

Both children,
r≠successor(x)