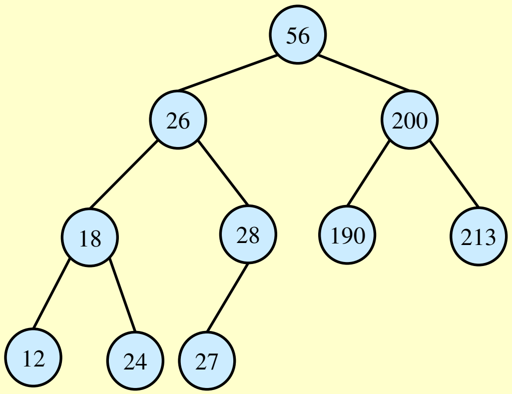
Binary Search Trees

Binary Trees

- Recursive definition
  1. An empty tree is a binary tree
  2. A node with two child subtrees is a binary tree
  3. Only what you get from 1 by a finite number of applications of 2 is a binary tree.

Is this a binary tree?

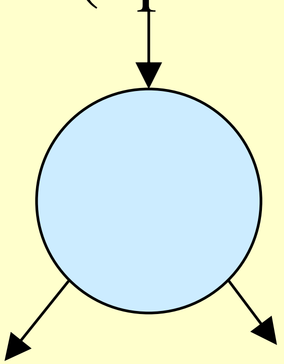


Binary Search Trees

- View today as data structures that can support dynamic set operations.
  - » Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- Can be used to build
  - » Dictionaries.
  - » Priority Queues.
- Basic operations take time proportional to the height of the tree – $O(h)$.

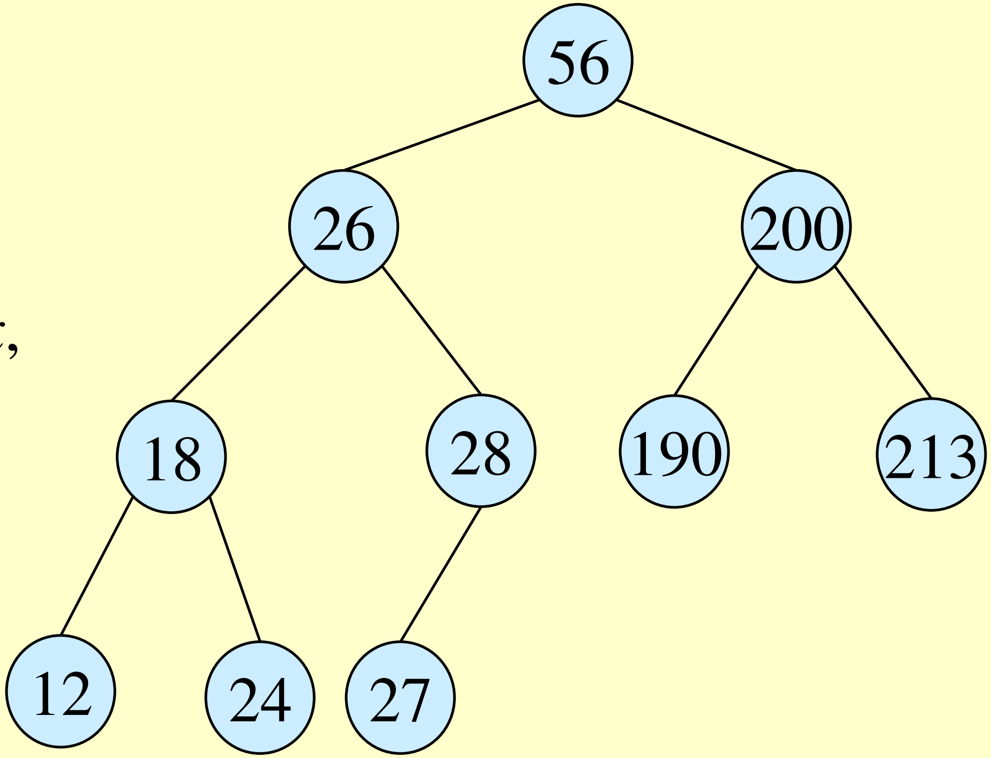
BST – Representation

- Represented by a linked data structure of nodes.
- *root*(*T*) points to the root of tree *T*.
- Each node contains fields:
  - » *key*
  - » *left* – pointer to left child: root of left subtree.
  - » *right* – pointer to right child : root of right subtree.
  - » *p* – pointer to parent. *p*[*root*[T]] = NIL (optional).



Binary Search Tree Property

- Stored keys must satisfy the *binary search tree* property.
  - » $\forall$ *y* in left subtree of *x*, then $key[y] \le key[x]$.
  - » $\forall$ *y* in right subtree of *x*, then $key[y] \ge key[x]$.

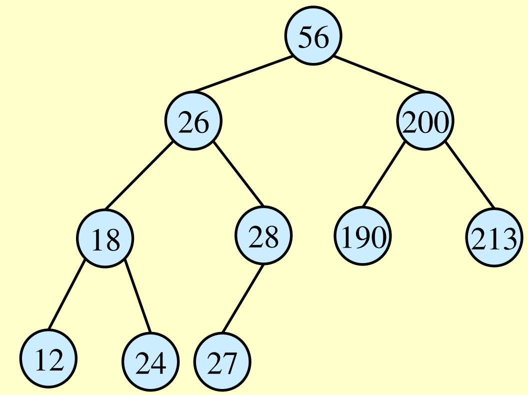


Inorder Traversal

The binary-search-tree property allows the keys of a binary search tree to be printed, in (monotonically increasing) order, recursively.

Inorder-Tree-Walk (x)

if $x \neq$ NIL
    then Inorder-Tree-Walk(*left*[*p*])
        print *key*[*x*]
        Inorder-Tree-Walk(*right*[*p*])



- How long does the walk take?
- Can you prove its correctness?

Correctness of Inorder-Walk

- Must prove that it prints all elements, in order, and that it terminates.
- By induction on size of tree. Size=0: Easy.
- Size >1:
  - » Prints left subtree in order by induction.
  - » Prints root, which comes after all elements in left subtree (still in order).
  - » Prints right subtree in order (all elements come after root, so still in order).

Querying a Binary Search Tree

- All dynamic-set search operations can be supported in $O(h)$ time.
- $h = \Theta(lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

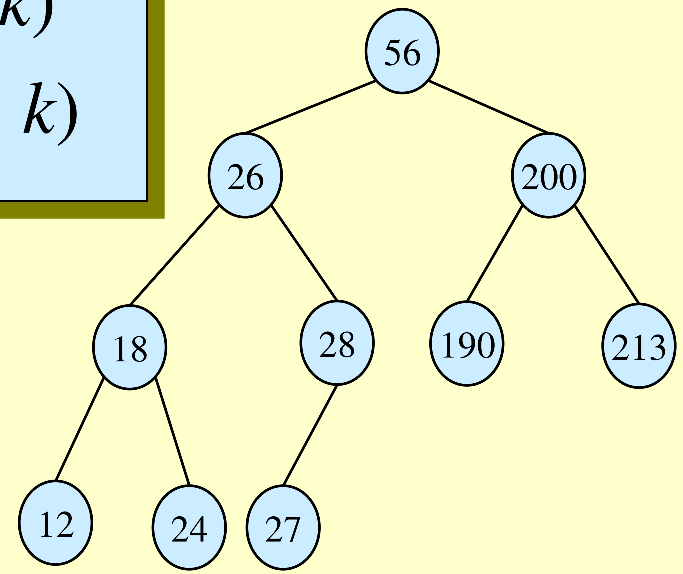
Tree Search

Tree-Search(x, k)

if x = NIL *or* k = *key*[x]
    then return x
if k < *key*[x]
    then return Tree-Search(*left*[x], k)
    else return Tree-Search(*right*[x], k)

Running time: $O(h)$

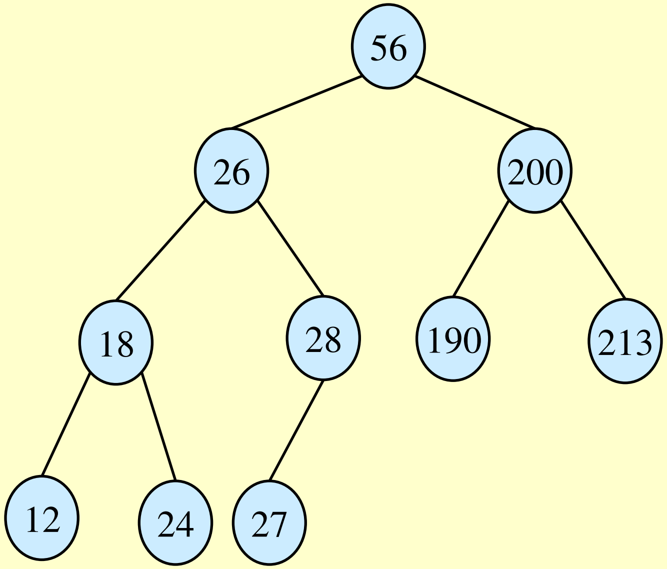
Aside: tail-recursion



Iterative Tree Search

Iterative-Tree-Search(x, k)

```
while x ≠ NIL and k ≠ key[x]
   do if k < key[x]
         then x ← left[x]
         else x ← right[x]
return x
```



The iterative tree search is more efficient on most computers.
The recursive tree search is more straightforward.

Finding Min & Max

- The binary-search-tree property guarantees that:
  - » The minimum is located at the left-most node.
  - » The maximum is located at the right-most node.

Tree-Minimum(x)

while $left[x] \neq NIL$
    do $x \leftarrow left[x]$
return x

Tree-Maximum(x)

while $right[x] \neq NIL$
    do $x \leftarrow right[x]$
return x

Q: How long do they take?

Predecessor and Successor

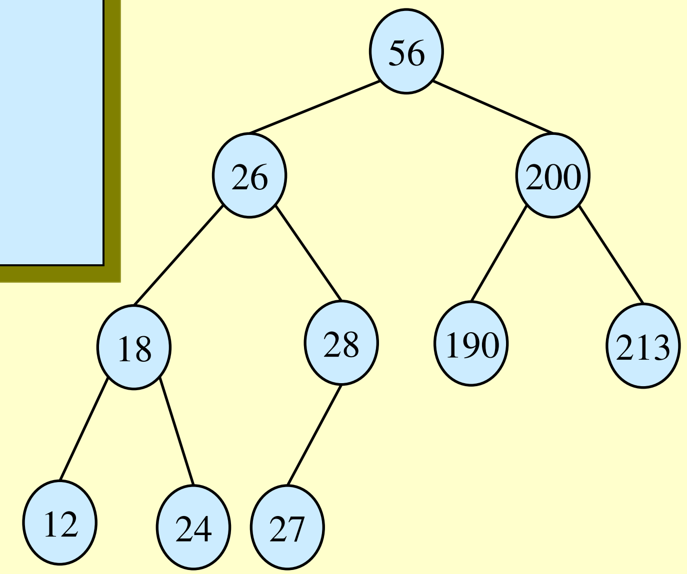
- Successor of node *x* is the node *y* such that *key*[*y*] is the smallest key greater than *key*[*x*].
- The successor of the largest key is NIL.
- Search consists of two cases.
  - » If node *x* has a non-empty right subtree, then *x*'s successor is the minimum in the right subtree of *x*.
  - » If node *x* has an empty right subtree, then:
    - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
    - *x*'s successor *y* is the node that *x* is the predecessor of (*x* is the maximum in *y*'s left subtree).
    - In other words, *x*'s successor *y*, is the lowest ancestor of *x* whose left child is also an ancestor of *x*.

Pseudo-code for Successor

```
Tree-Successor(x)
    if right[x] ≠ NIL
        then return Tree-Minimum(right[x])
    y ← p[x]
    while y ≠ NIL and x = right[y]
    do x ← y
        y ← p[y]
    return y
```

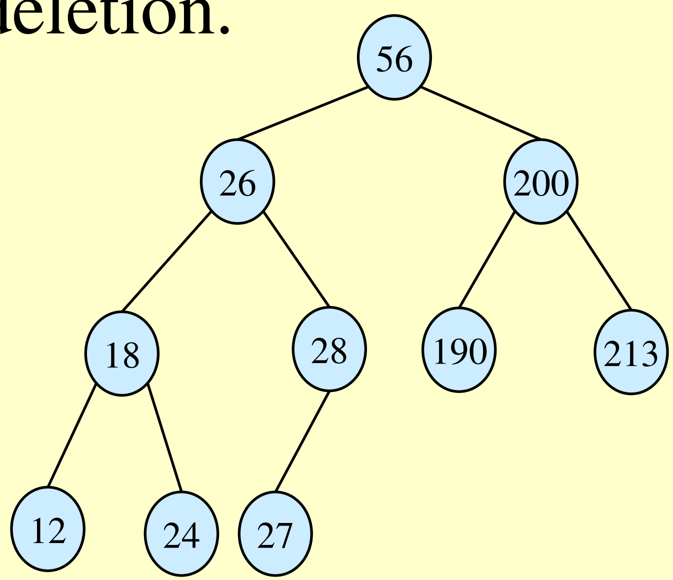
Code for ***predecessor*** is symmetric.

Running time: $O(h)$



BST Insertion – Pseudocode

- Change the dynamic set represented by a BST.
- Ensure the binary-search-tree property holds after change.
- Insertion is easier than deletion.



Tree-Insert(*T*, *z*)

```
y ← NIL
x ← root[T]
while x ≠ NIL
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = NIL
    then root[t] ← z
    else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z
```

Analysis of Insertion

- Initialization: $O(1)$
- While loop in lines 3-7 searches for place to insert z, maintaining parent y.
  This takes $O(h)$ time.
- Lines 8-13 insert the value: $O(1)$

$\Rightarrow$ TOTAL: $O(h)$ time to insert a node.

Tree-Insert(T, z)

```
1.  y ← NIL
2.  x ← root[T]
3.  while x ≠ NIL
4.      do y ← x
5.          if key[z] < key[x]
6.              then x ← left[x]
7.              else x ← right[x]
8.  p[z] ← y
9.  if y = NIL
10.     then root[t] ← z
11.     else if key[z] < key[y]
12.         then  left[y] ← z
13.         else right[y] ← z
```

Exercise: Sorting Using BSTs

Sort (A)

for $i \leftarrow 1$ to n

do tree-insert($A[i]$)

inorder-tree-walk(*root*)

- » What are the worst case and best case running times?
- » In practice, how would this compare to other sorting algorithms?

Tree-Delete (T, x)

if x has no children ♦ case 0

then remove x

if x has one child ♦ case 1

then make $p[x]$ point to child

if x has two children (subtrees) ♦ case 2

then swap x with its successor

perform case 0 or case 1 to delete it

⇒ TOTAL: $O(h)$ time to delete a node

Deletion – Pseudocode

Tree-Delete(T, z)

/* Determine which node to splice out: either z or z's successor. */

- **if** *left*[z] = NIL **or** *right*[z] = NIL
- **then** $y \leftarrow z$
- **else** $y \leftarrow$ Tree-Successor[z]

/* Set x to a non-NIL child of x, or to NIL if y has no children. */

if *left*[y] $\neq$ NIL
    then $x \leftarrow$ *left*[y]
    else $x \leftarrow$ *right*[y]

/* y is removed from the tree by manipulating pointers of $p[y]$ and x */

if $x \neq$ NIL
    then $p[x] \leftarrow p[y]$

/* Continued on next slide */

Deletion – Pseudocode

Tree-Delete(T, z) (Contd. from previous slide)

if $p[y]$ = NIL
    then $root[T] \leftarrow x$
    else if $y \leftarrow left[p[i]]$
        then $left[p[y]] \leftarrow x$
        else $right[p[y]] \leftarrow x$

/* If z's successor was spliced out, copy its data into z */

if $y \neq z$
    then $key[z] \leftarrow key[y]$
          copy y's satellite data into z.
return y

Correctness of Tree-Delete

- How do we know case 2 should go to case 0 or case 1 instead of back to case 2?
  - » Because when *x* has 2 children, its successor is the minimum in its right subtree, and that successor has no left child (hence 0 or 1 child).
- Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating lopsided tree.

Binary Search Trees

- View today as data structures that can support dynamic set operations.
  - » Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- Can be used to build
  - » Dictionaries.
  - » Priority Queues.
- Basic operations take time proportional to the height of the tree – $O(h)$.

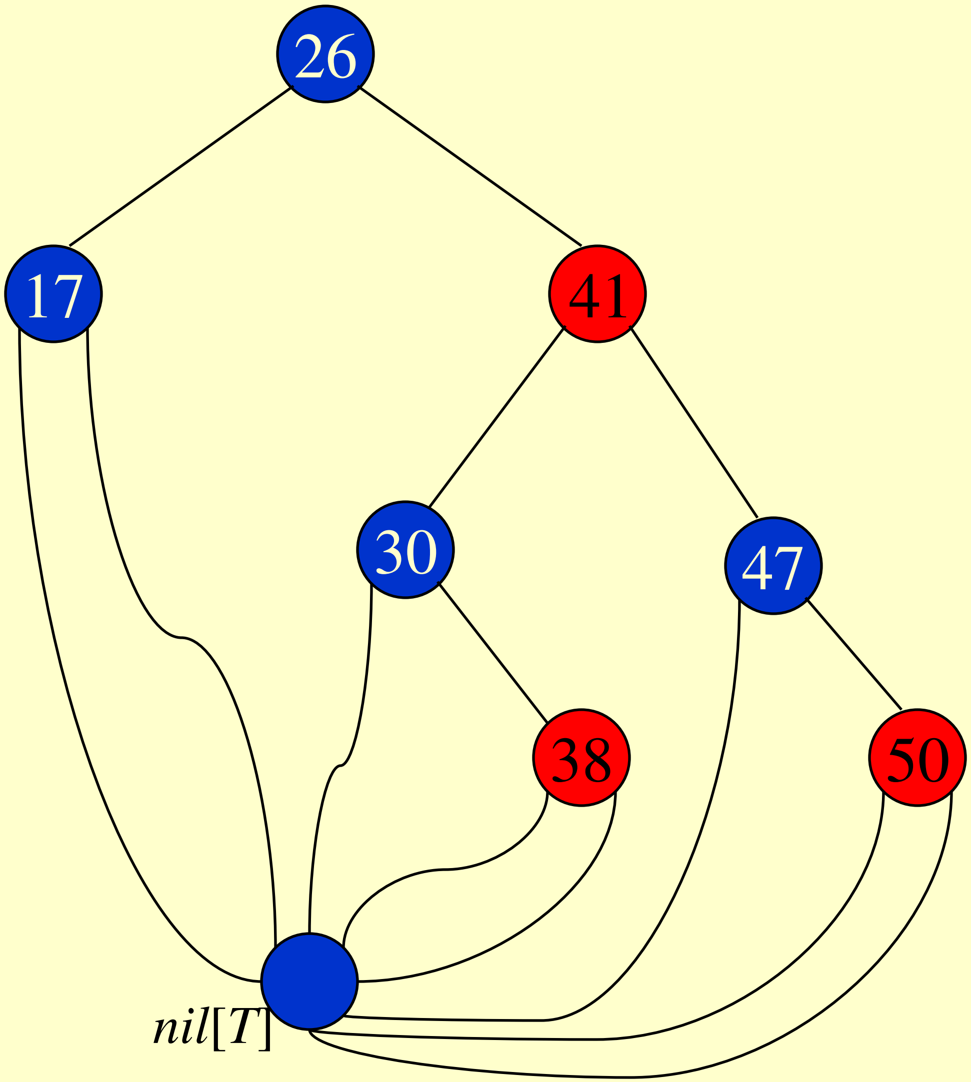
Red-black trees: Overview

- Red-black trees are a variation of binary search trees to ensure that the tree is ***balanced***.
  - » Height is $O(\lg n)$, where n is the number of nodes.
- Operations take $O(\lg n)$ time in the worst case.

Red-black Tree

- Binary search tree + 1 bit per node: the attribute *color*, which is either **red** or **black**.
- All other attributes of BSTs are inherited:
  - » *key*, *left*, *right*, and *p*.
- All empty trees (leaves) are colored black.
  - » We use a single sentinel, *nil*, for all the leaves of red-black tree *T*, with *color*[*nil*] = black.
  - » The root's parent is also *nil*[*T*].

Red-black Tree – Example



Red-black Properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf (*nil*) is black.
4. If a node is red, then both its children are black.

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

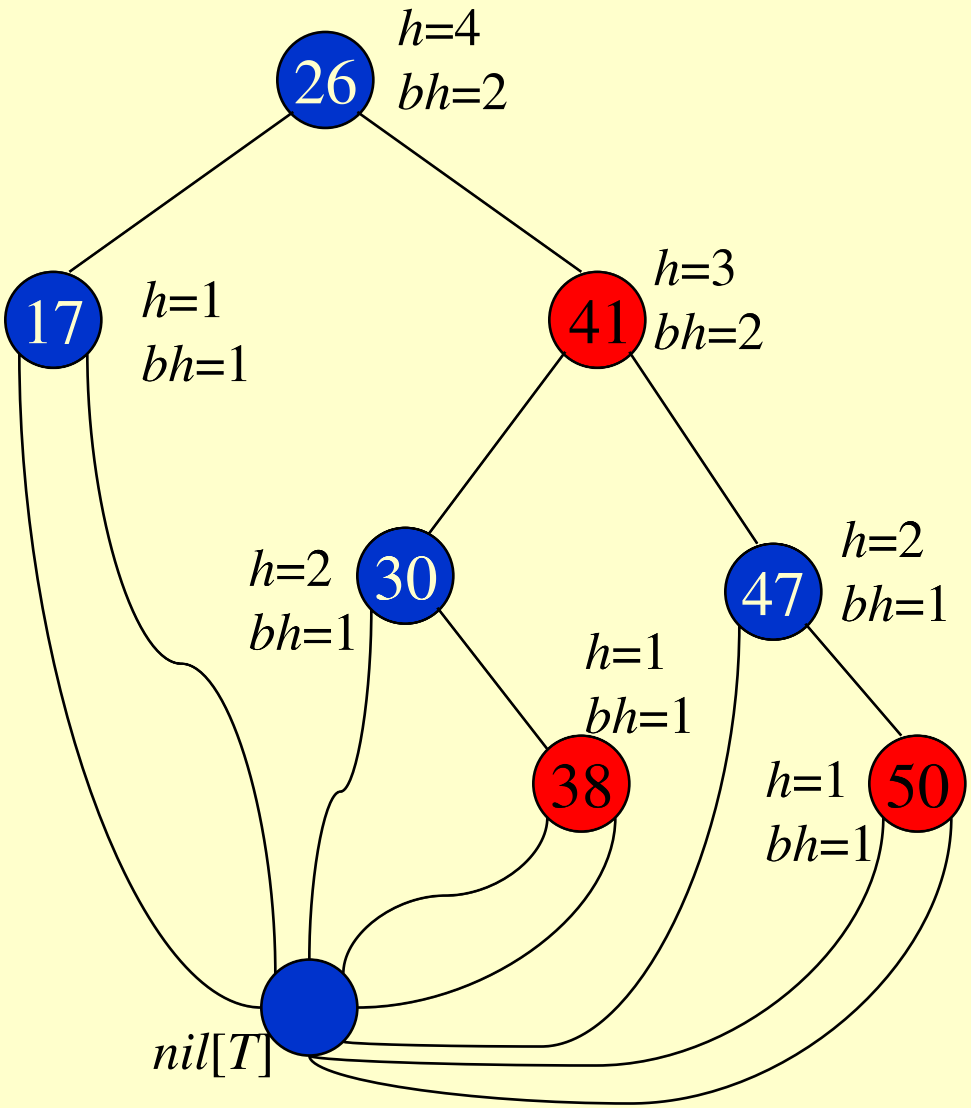
Height of a Red-black Tree

- Height of a node:
  - » Number of edges in a longest path to a leaf.
- Black-height of a node x, $bh(x)$:
  - » $bh(x)$ is the number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x.
- Black-height of a red-black tree is the black-height of its root.
  - » By Property 5, black height is well defined.

Height of a Red-black Tree

- Example:

- Height of a node:
  - » Number of edges in a longest path to a leaf.
- Black-height of a node $bh(x)$ is the number of black nodes on path from x to leaf, not counting x.



Hysteresis : or the value of lazyness

- **Hysteresis**, n. [fr. Gr. to be behind, to lag.]
  a retardation of an effect when the forces acting upon a body are changed (as if from viscosity or internal friction); *especially*: a lagging in the values of resulting magnetization in a magnetic material (as iron) due to a changing magnetizing force